Enhancement of Query Processing on XML Data

Yang Rui

NATIONAL UNIVERSITY OF SINGAPORE
2006

Enhancement of Query Processing on XML Data

Yang Rui
(Master of Engineering)
(North China Electric Power University, China)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPY
DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE
2006

Acknowledgements

"Many a little makes a mickle". The work of this thesis is based on the cooperation of many people. I would like to take this opportunity to express my gratitude to all those who gave me the possibility to complete this thesis.

I want to thank the Computer Science Department of National University of Singapore for providing scholarship to me and for giving me permission to commence this thesis, to do the necessary research work and to use departmental facilities.

I am deeply indebted to my supervisor Dr. Anthony Tung, for his stimulating suggestions and encouragement which helped me in all the time of research for and writing of this thesis. He took me on the process of learning and made himself available even through his very heavy travel, work and teaching schedule. At the same time, I would also like to gratefully acknowledge the support of some very special individuals. They are Professor Tok Wang Ling, Dr. Panos Kalnis and Dr. Stephane Bressan. I worked with them to finish the papers and reports which consist of the main part of this thesis. Thanks for their patience and directions.

My former colleagues from the computational biology lab and database/e-commerce lab supported me in my research work. Special thankfulness should be expressed to Dr. Jiaheng Lu. They mirrored back my ideas, an important process for me to shape my thesis paper and future work. Also, we shared the enjoyable working environment, interesting lectures and seminars; I appreciate their cherishable friendship.

Finally, I wish to express my love and gratitude to all my family and friends. I'd particularly like to thank my parents and brother for never advising me to quit this project. They had more faith in me than could ever be justified by logical argument. Their endless support, encouragement, and understanding is my motive power to finish the long journey in obtaining my degree in Computer Science.

Contents

Summary

XML documents have recently become ubiquitous because of their varied applicability. It is believed that progressively more and more Web data will be in XML format. Communities of business and sciences are defining their own DTD to provide for a uniform representation of data in specific areas [85, 87, 64, 62]. For example, in business, the efforts have been taken to develop standardized XML vocabularies for recruiting and other human resource functions [51], for publishers and printers (XPP) [42] etc. In scientific area, especially the biological [81, 64] and chemistry area [63, 82], researchers have brought XML power to the management of scientific data. The initial impetus for XML may have been primarily to enhance the ability of remote applications to interpret and operate on documents fetched over the Internet. However, from a database point of view, XML raises different exciting possibility: with data stored in XML documents, one should be able to issue queries over sets of XML documents to extract, synthesize, and analyze their contents. Given the broad adoption of XML, it pressed for efficient manipulations on the XML data in huge dataset. In this thesis, the efficient similarity query processing and pattern query processing on XML data is extensively studied.

XML data is self-describing through the nested structures of elements. Therefore, XML data are usually modeled as rooted, ordered, labeled trees. Similarity search is to find all objects in the database which are within a given distance from a given object (range query) or to find the k most similar objects in the database which are closest in distance to a given object (k-NN query). Although similarity search has been extensively studied on multivariate numeric data and categorical data vector, searching for similar trees is still an open problem due to the high complexity of computing the tree edit distance. In this thesis, XML data is transformed into an numerical multidimensional vector which encodes the original structure information and content information. The L_1 distance of the corresponding vectors, whose computational complexity is linear to the data size, forms a lower bound for the edit distance between trees. Based on the theoretical analysis, a novel algorithm is presented which embeds the proposed distance into a filter-and-refine framework to process similarity search on tree-structured data. The experimental results show that the new algorithm reduces dramatically the distance computation cost. And it is especially suitable for accelerating similarity query processing on large trees in massive datasets.

For the XML pattern query processing, an important operation is to search for all occurrences of a twig pattern in an XML database. Most of the existing research work surprisingly output all the distinct matches for all query nodes. However, in practice, queries written in XPath or XQuery only require to output answers which consist of the distinct matches to the selected query nodes (called distinguished nodes). The straightforward approach is to makes an appropriate projection on the selected node matches by post-processing the outputs of previous methods. Obviously, it is not optimal in most cases. At the same time, the previous approaches are optimal only for limited class of queries. In this thesis, we prove that the sub-optimality of prior algorithms is due to the matching blocks in the data streams. However, if only bindings of the distinguished nodes are required, most blocks can be conquered by caching limited number of elements in the main memory (bounded by the depth of documents). Based on these theoretical analyses, two efficient query processing algorithms named TwigContainment and TwigPrefix are proposed. They utilize containment labeling and prefix labeling respectively.

Unlike the prior methods, these algorithms only take one phase to avoid outputting irrelevant intermediate path solutions. Moreover, these two algorithms identify the same optimal class which is much larger than those identified by the previous approaches. Finally, a set of experimental results on both real-life datasets and synthetic datasets verify the effectiveness and the optimality of our new algorithms.

In summary, the contribution of this thesis is that we have successfully provided efficient solutions to two types of similarity queries - the range query and the k-NN query, and pattern queries on XML data. The results of our experiments also suggest that our methods are especially suitable for accelerating the query processing on the massive datasets consisting of XML data of large size and deeply-nested elements with infrequent updates.

List of Tables

List of Figures

Chapter 1

Introduction

Internet and Web application is becoming more and more important nowadays. Therefore, the publication of electronic data has been becoming universal. Most of these electronic data appear as HTML documents on the Web and are generated automatically from database. However, HTML aims to specify the representation of the information instead of the structure and content of it. So, although HTML document is readable to human-beings, it is difficult for other application programs to understand such data. XML (eXtensible Markup Language) [19] was proposed by the World Wide Web Consortium (W3C) as a new standard for data exchange on the Web to complement HTML. Unlike HTML, XML is a textual representation of data which utilize the nested tree hierarchy to depict the structural relationship between the data components. Figure 1.1 is a fragment of a XML document which describe the movie information.

The basic component in XML data is the element, i.e., a piece of text bounded by matching tags (such as <movie> and </movie> in the Figure 1.1). The elements can be nested. Each element can be either of atomic value (i.e., raw character data) or composite value (i.e., a sequence of nested subelements). In Figure 1.1, the root element ($MovieDB$) has three nested subelement ($movie$, $director$ and $actor$). The order of the subelements within an element is sometimes significant in XML document (e.g. the order of the actors). It is allowed to associate attribute/value pairs with elements (e.g., the

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE $W4F\_DOC$ SYSTEM "movies.dtd">
<MovieDB>
    <Movie id = "a885", language = "English" >
        <Title> Night of the Hunter, The </Title>
        <Year> 1955 </Year>
        <Genres>
            <Genre> Drama </Genre>
            <Genre> Thriller </Genre>
        </Genres>
        <Director $director_{id}$ = "a133"> Charles Laughton </Director>
        <Cast>
            <Actor $actor_{id}$ = "a735"> Robert Mitchum </Actor>
            <Actor $actor_{id}$ = "a459"> Shelley Winters </Actor>
            ...
        </Cast>
    </Movie>
    ...
    <Director id = "a133">
        <FirstName> Charles </FirstName>
        <LastName> Laughton </LastName>
        <movie $movie_{id}$ = "a8904885"/>
        ...
    </Director>
    ...
    <Actor id = "a735">
        <FirstName> Robert </FirstName>
        <LastName> Mitchum </LastName>
        <movie $movie_{id}$ = "a885"/>
        ...
    </Actor>
    ...
</MovieDB>
```

Figure 1.1: An Example of XML Data

$language$ specification of the $movie$ in the above example). A distinct attribute is object IDs (e.g., the ID attributes of the $movie$, $actor$ and $director$ elements). And through this attribute and attribute IDREF (e.g., the $movie_{id}$ attribute of the $movie$ element under $actor$ and $director$), XML allows the reference between elements. Attributes should be unique among each element. The part of the syntax not enclosed within brackets is referred to as PCDATA (Parsed Character Data). We say a document is well-formed if it satisfies all these constraints. More details on the XML specification can be found in [19]. We can see that XML is self-describing and irregular. In XML, new tags may be defined at will to specify information and the structure relationship between information elements. And the structure can be nested to arbitrary depth. And an XML document can contain an optional description of its grammar. It is widely recognized as the data representation, exchange and integration standard of the future.

Given the broad adoption of XML, a database system is required for efficient manipulation of XML data. In previous research efforts, XML database has been implemented by using either traditional file system [3], relational database system [98, 38, 41], object-oriented database system [15, 59, 100, 117] or semi-structured database system [21, 78, 45, 6]. The native XML databases have been implemented as well [78, 6, 104, 103, 40, 52] (Accordingly, the other implementation mentioned above can be called XML-enabled database). Using a file system is straightforward. However, it does not support complex query processing (Full text searches are obviously not accurate since markup, text and other syntax component not be distinguished.). Relational database implementation is regarded as practical approach due to its wide deployment in commercial world and its mature RDBMS technologies, e.g.,indexing, concurrency control and transaction management, can be well exploited. Object-oriented database systems allow a flexible storage system of XML data and support complicated query processing. However, both of them are based on rigid schema definition and are not natural for modeling the irregular XML data relationship. Furthermore, object-oriented database systems are neither mature nor efficient enough for industry adoption. From the above example, we can see that XML data are similar to semi-structured data. Both of them are self-describing and have no rigid structure. So some research works done on semi-structured data can be extended to process XML data. But there are still some differences between them and XML data: XML is ordered while semi-structure data is not; XML can mix text and element together; and XML have a lot of other stuff: entities, processing instructors and comments. These differences make XML data management harder than semi-structured data. Native XML database systems are designed especially to store XML documents. Like other databases, they support features like transactions, security, multi-user access, programmatic APIs, query languages, and so on. Native XML database is capable to reserve the proper characteristics of XML. In addition, it can handle schema changes and data updates more easily. However, efficient data manipulations are required for this kind of specialized database. This inspires the research work of this thesis.

The efficiency problem of managing and querying XML documents poses interesting challenges for database researchers. There are a lot of literatures about XML query language [11], XML query optimization [79, 94, 98, 46, 58, 112, 7, 30] (including XML numbering/encoding scheme, XML indexing, XML summary analysis etc.), and XML compression [108, 70]. However, little research work has been done on the XML data processing based on similarity measurement. And for the pattern query, optimizing the I/O cost and reducing the size of the intermediate results still appeal lots of attentions. The work of this thesis is mainly focused on improving the similarity query (or similarity search) and pattern query (or pattern search) processing on XML data. In the next three sections, we give a brief introduction to the modeling of XML data, the similarity search and pattern search on XML. In the last 4 sections, we also present the motivation, main contribution and organization of this thesis.

1.1 XML Data Model

Two types of models are most frequently used for XML data. One is the Stanford's Object Exchange Model (OEM) [89, 4, 78]. Another one is the W3C's Document Object Model (DOM) [94, 58].

OEM was introduced in TSIMMIS (The Stanford-IBM Manager of Multiple Information Sources) as a self-describing way of representing metadata. OEM was later modified for use in the Lore (Lightweight Object Repository) system to represent semistructured data. In the Lore scheme, each object consists of a object identifier (oid), a type and a value. These effectively represent relationships between the containing object and the target object. In order to make the OEM model suitable for XML data, the author of [32] made some modification to it: XML element is a pair $(eid,\ value)$; where eid is an unique element identifer, and $value$ is either an atomic text string or a complex value containing (optionally) the following four components: string-valued tag, an ordered list of attribute-value pairs, an ordered list of attributes of type IDREF or IDREFS in the form $(label,\ eid)$, where ***label*** is the attribute name, and an ordered list of subelements in the form $(label,\ eid)$, where the ***label*** is the subelement tags. Figure 1.2 is the OEM model for the movie element of the XML document fragment in Figure 1.1.

DOM model provides a mechanism for programs to access and manipulate parsed XML content as a collection of objects. DOM represents a document as a hierarchy of objects, called nodes, which are derived (by parsing) from a source representation of the document. The DOM Level 1 working draft defines a set of object classes (and their inheritance relationships) for representing documents: document, element, attribute, text, PI (processing instructor), comment and namespace objects. The XML document is presented to an application as a collection (actually, a tree) of objects. Most of these objects would be of type node, and specifically of its subtypes element (representing the individual elements) and text (representing the content). Figure 1.3 is the tree representation of

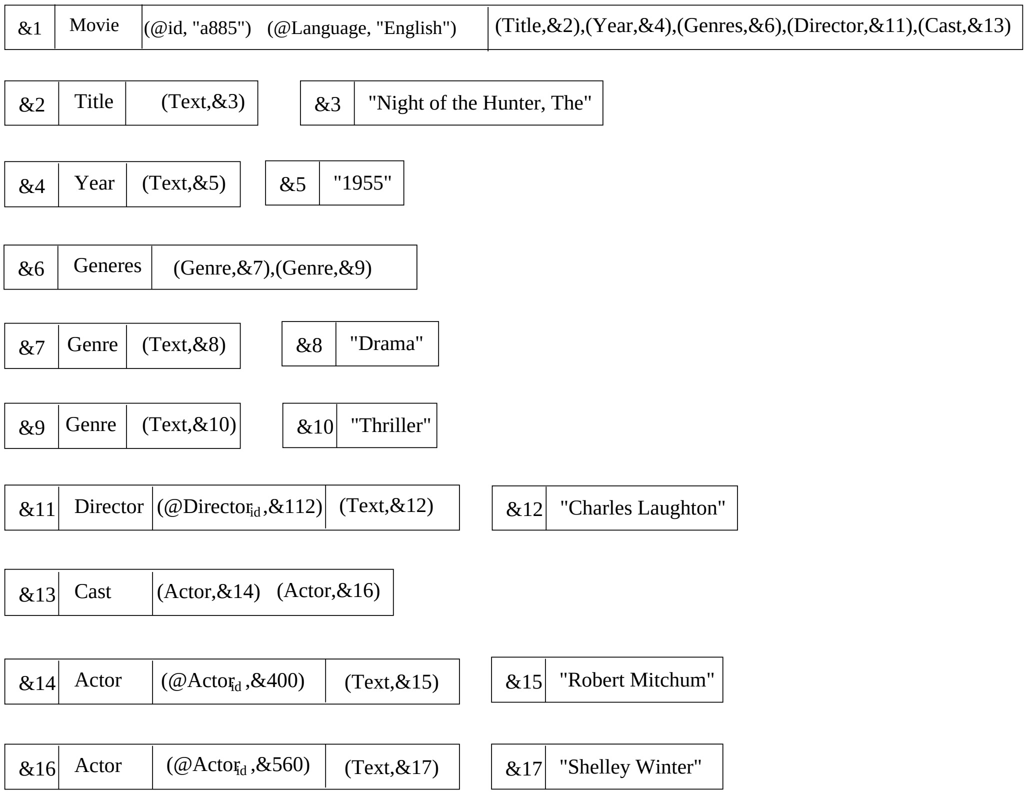

&1 Movie (@id, "a885") (@Language, "English") (Title,&2),(Year,&4),(Genres,&6),(Director,&11),(Cast,&13)
&2 Title (Text,&3)
&3 "Night of the Hunter, The"
&4 Year (Text,&5)
&5 "1955"
&6 Generes (Genre,&7),(Genre,&9)
&7 Genre (Text,&8)
&8 "Drama"
&9 Genre (Text,&10)
&10 "Thriller"
&11 Director (@Director$_{id}$,&112) (Text,&12)
&12 "Charles Laughton"
&13 Cast (Actor,&14) (Actor,&16)
&14 Actor (@Actor$_{id}$,&400) (Text,&15)
&15 "Robert Mitchum"
&16 Actor (@Actor$_{id}$,&560) (Text,&17)
&17 "Shelley Winter"

Figure 1.2: An OEM Model of XML Data Structure

the DOM model of the above example. (The nodes are labeled in abbreviated form and the text nodes are ignored for clarity.)

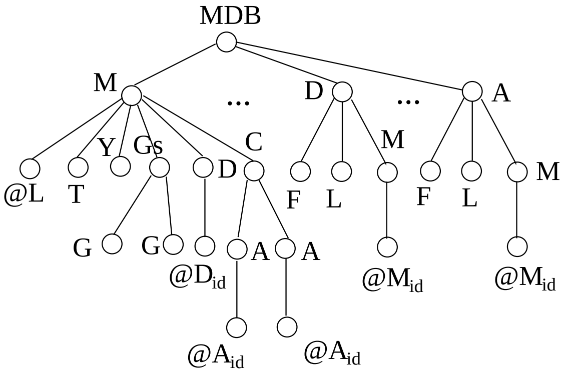


Figure 1.3: The Tree Representation of DOM Model of XML Data

In order to research the characteristics of XML data, we need the formalized data model. In this thesis, XML database is modeled as a collection of rooted, ordered, labeled trees, denoted as D. As shown above, the XML documents may have hyperlinks to other documents. In the meanwhile cycles may exist in the data due to the ID, IDREF attributes of elements. Including these in the model gives rise to a graph rather than a tree. However, they are not important in terms of the structures of the documents considered in this thesis. Hence, the ID-references and hyperlinks are ignored for simplicity. Each XML data is modeled as a rooted, ordered, labeled tree T. There exists only one root note, which has no parent. Every other node of the tree has exactly one parent and it can be reached through a path of edges from the root. A tree T is called *labeled tree* if each node is assigned a symbol from a fixed finite alphabet. For XML data, the alphabet consists of all the tag names and attribute names of XML data. And a tree is called *ordered tree* if a left-to-right order among siblings in T is given and order counts during data processing. It is obvious that the graphic representation of our model is similar to that of DOM except that we focus on the structural information which consists of the relationships between elements and between elements and attributes. The notations related to the data model is given in Chapter 2.

1.2 XML Similarity Search

Similarity search is an important core operation for many data analysis tasks on multimedia and time-series databases, biological and scientific databases. In this thesis, I focus on two typical kinds of similarity queries on XML data: *range query* and *k-Nearest-Neighbor query* (k-NN query for short). Range queries find all objects in the database which are within a given distance τ from a given object; k-NN queries find the k most similar objects in the database which are closest in distance to a given object. Other types of search can be composed by these two similarity queries. These problems have been extensively studied on numerical multi-dimensional data [50, 97, 13, 14, 72, 93, 119] and the distance measures depend on the order among data. However, in many other applications, multivariate analysis is applied on complex data domains which may not have a natural order. Transaction data (or categorical data) is an example of such domain. In recent years, several indexing approaches were proposed to address the similarity search problem on transaction datasets [8, 83, 77] too. XML data is another example among which there are no natural orders.

XML data are often with no schema specification. Even if there is a schema, the data conforms to it flexibly. Elements and attributes can be optional and one type of elements can occur multiple times. Furthermore, in the XML document, the semantics specified implicitly by the relationship between its components. Then the tree structures play an important role on differentiating data. The measurement of XML data similarity can be precise only if this information is exploited and introduced into the measure function. Thus, the traditional distance measurements cannot be used straightforward in this area. So it is still an open problem. Since XML data are usually modeled as rooted, ordered, labeled trees, and due to the flexibility of XML representation power, several existing works employ the tree edit distance measure on the XML data trees, i.e., the minimum number of operations required to transform one tree to the other. The definition of allowable tree edit operation varies according to the application [9, 86, 49, 125, 126, 105, 124]. However, the computation complexity of this distance measure is quite high. In Chapter 2, a brief introduction of these measures will be given. Assuming a similarity measure between XML data, $Dist(T,\ T')$, the formal definition of similarity queries are give in Definition 1.2.1, Definition 1.2.2 respectively.

Definition 1.2.1 (k-NN query). A k-NN query $Q_k = \langle Q, k, D\rangle$ retrieves a set R_k of k data from Dataset D, such that for any two data $T \in R_k$, $T' \notin R_k$, $Dist(Q,T) \leq Dist(Q,T')$.

Definition 1.2.2 (Range query). A range query $Q_r = \langle Q, \varepsilon, D\rangle$ retrieves a set of data R_r

from Dataset D, such that $\forall T \in R_r, Dist(Q,T) \leq \varepsilon$; and $\forall T' \notin R_r, Dist(Q,T') > \varepsilon$.

1.3 XML Pattern Query

Unlike the similarity query, the pattern query on XML data should not be processed by measuring the similarity between the query pattern and the XML data straightforwardly. Instead, pattern queries specify both the structural and value constraints the result portions of XML document should satisfy. As for the basic query abstractions, the XML query language should support both select operation and join operation. Select operation picks up the elements satisfying the constrains specified in the query, while join condition compares two or more XML attributes or data belonging to the same XML data or different documents. Additionally, when dealing with XML data in which the exact structure is not known, it is convenient to use a form of ”navigational” query based on path expressions which uses wildcards and regular expressions. Various query languages for extracting, transforming and integrating the XML content have been defined: Lorel [4], XQuery [2, 37] XML-QL, XML-GL, XSLT, XQL and Quilt [11, 23]. Some of them are in the tradition of database query languages like SQL, OQL and Datalog, while others are more closely inspired by XML.

```
FOR          $\$t_0$ IN doc(“movies.xml”)/movieDB//movie[@Language = “English”],
WHERE        $\$t_0//Director =$ “Charles Laughton”,
AND          $\$t_0//Actor =$ “Robert Mitchum”,
ORDER BY     $\$t_0/Title,$
RETURN       $<$ Movie $>$  $\{\$t_0/Title\}$  $<$ /Movie $>$
```

Figure 1.4: An Example of XQuery

XQuery is defined by the W3C and is supported by all the major commercial database engines (IBM, Oracle, Microsoft, etc.). In this thesis, we use it as the query language of XML. XQuery is for finding and extracting elements and attributes from XML documents. It is built on XPath [1] expressions which navigate through elements and attributes in an XML document. The Syntax of XPath is defined as:

$$\begin{aligned} PathExpr &::= /step_1/step_2/\cdots/step_n; \\ step &::= Axis :: NodeTest\ \ Predicate* \end{aligned} \tag{1.1}$$

Each XPath expression consists of a sequence of location steps. Each step contains the Axis, the NodeTest specification and zero or more Predicates. Axis specifies the tree relationship between the nodes selected by the location step and the context node. NodeTest prescribes the node name or node type selected by it. And Predicates are expressions in square brackets, which further refine the set of nodes selected by the location step. XPath has 13 different axes of navigation, i.e. ancestor, ancestor-or-self, parent, attribute, child, descendant, descendant-or-self, self, following, following-sibling, preceding, preceding-sibling and namespace. In this thesis, we mainly study the child and descendant axes navigation which are used to traverse to a child or a descendant element respectively. They can be represented by '/' and '//' respectively for abbreviation. Figure 1.4 shows an XQuery example. The $doc()$ function is used to open the "movies.xml" file and specify the context. The path expression $doc(“movies.xml'')/movieDB//movie$ is used to select all the movie elements under $movieDB$ in the "movie.xml" file. All the selected elements are bound with the variable $\$t_0$ (An XQuery variable is defined with a $\$$ followed by a name, e.g. $\$t_0$). The predicate $[@language = “English'']$ further constrain that the selected movie are in English. Symbol @ followed by the name is used to retrieve the attribute.

XQuery also uses FLWOR expressions. FLWOR is an acronym for "FOR, LET, WHERE, ORDER BY, RETURN". In Figure 1.4, the FOR clause selects all movie elements under the document element that satisfy the query conditions and combines them with the variable $\$t_0$. The WHERE clause specify the selection condition, i.e., the director is “Charles Laughton” and one of the actors is “Robert Mitchum”. The ORDER BY clause requires that the results will be sorted by the $title$. And the RETURN clause specifies what should be returned, i.e., the $title$ elements which satisfy the predicate condition, and constructs the resulting movie elements.

As shown in the previous example, XQuery specify the pattern of selective predicate on multiple elements which satisfy the specified tree structural relationship. Thus, these queries are also called structural queries. The most frequently proposed XML structural queries are tree (twig) pattern queries which can be represented by a node-labeled tree [20]. For example, the following XQuery expression in Equation 1.2 can be represented by the twig shown in Figure 1.5.

$$\begin{gathered}\text{//Movie[@Language = ‘English’ AND ./Director = “Charles Laughton”} \\ \text{AND .//Cast/Actor = “Robert Mitchum”]/Title}\end{gathered} \tag{1.2}$$

Since both XML data and XML queries are represented as trees, in the rest of the

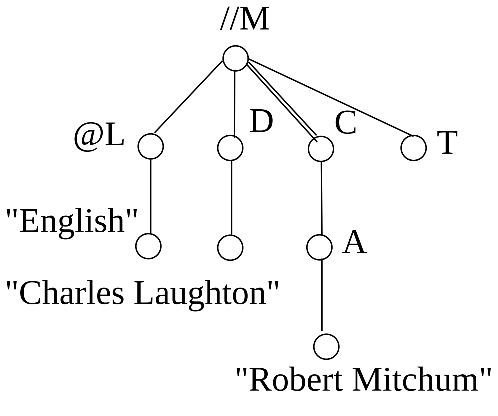


Figure 1.5: The Twig Pattern Query

thesis, “node” is used to refers to a tree node in the twig pattern, while “element” refers to an element in the dataset, when the discrimination is necessary. Each node in the twig also represents the content predicates on it, which usually specify tag names of the elements, attribute value comparison, and string values of elements. The edges between the nodes depict the structural containment relationships between the nodes. The parent-child relationship predicates (PC for abbreviation) between elements and the element-attribute constrains are represented by the single lines, while the ancestor-descendant relationship predicates (AD for abbreviation) are represented by the double lines.

Evaluating a XML twig pattern query Q_p on a XML database D is to identify all the matches of the query nodes in D. A match of Q_p in D is actually a mapping from the query nodes to the elements (or other components like attributes) of a certain XML data T such that:

1. The predicates specified by the query nodes can be satisfied by their respective images under the mapping to T;

2. The structural relationship depicted by the edges between query nodes can be satisfied by their respective images under the mapping to T.

According to [20], the answer to Q_p can be modeled as a n-ary relation $(d_1,\ d_2,\ \cdots,\ d_n)$ where each tuple is a mapping of the query nodes and n is the number of query nodes, i.e., the size of the query Q_p, denoted as $|Q_p|$.

In recent years, many methods have been proposed to match XML twig queries efficiently. These methods can be classified into three categories according to the searching strategies: the relational-based methods [98, 38, 41, 18], the path navigation methods [46, 80, 58, 32] and the structure-join-based methods. The structure join methods can be further classified into binary structure join [41, 79, 10, 104, 103, 98, 123] and holistic twig join methods [20, 28, 74, 55]. The relational-based methods require mapping the XML data and store them into relational database, transforming the queries proposed in XQuery into SQL and constructing the results retrieved from relational database into XML documents according to query specification. As mentioned above, the relational-based methods make use of the high reliability, scalability and optimized performance of relational database. However, the challenge is that there is mismatch between the relational model and that of XML. The relational model is normalized, flat and fragmented, while XML is un-normalized, nested and monolithic. These lead to the limitations of the relational implementation of XML database. The path navigation methods are based on the structural summary or path expression index and speed up query evaluation on XML data by restricting the search to only relevant portion of the XML data.[1] The structure join methods are also utilized as the core operation to answer queries. Various element positional numbering schemes are devised to identify the elements which satisfy the structural predicates [35, 123, 107, 88, 74]. Binary structure join methods decompose the query pattern into a set of binary structural predicates and each predicate is evaluated separately. By "stitching" together the binary structure join results, the final answers of the whole queries can be obtained. Indexes can be utilized to accelerate the binary structure join process. However, there may exist too many intermediate results which cannot contribute to the final answers. The suboptimality is incurred by query decomposition. Unlike binary structure join approaches, the family of holistic twig join methods try to process the queries as a whole and make sure that each output partial answer to the path pattern queries can be merge-joinable with at least one partial answer for each other path pattern in the twig. All these methods are introduced in Chapter 2.

1.4 Motivation for Similarity Query Study

Just as the management of traditional types of data, many research disciplines are based on the similarity measurement of XML data, such as schema extraction, XML data storage and retrieval, XML data version management, and the data mining techniques like nearest neighbor classification methods, cluster analysis etc. And similarity search is an important core operation for many data analysis tasks on multimedia and time-series

[1] Some of the path expression index are proposed to be implemented in relational database.

databases, biological and scientific databases. Now that more and more data are conveyed in XML language, efficient processing of this type of queries is a pressing requirement.

The straightforward solution to similarity search is to sequentially scan all the data items in the database. However, such processing is not practical at all. Firstly, with the fast development of bioscience and the wide employment of internet database, the volumes of the available complex data are becoming larger and larger. The size of a gene sequence file is usually several Gigabytes. It is unacceptable to load all data into the main memory to sequentially scan such large volumes of data. Secondly, the computational complexity of the distance measure between XML data makes it prohibitive for bulk operations in the database. As mentioned in Section 1.1, XML data are modeled as rooted ordered labeled trees. The well known distance function for trees is the edit distance, which is defined as the minimum number of tree edit operations required to transfer one tree into another. To compute this distance, dynamic programming method is often used and the best known tree edit distance evaluation algorithms have more than $O(n^2)$ runtime and space complexity for ordered trees with n nodes [125, 29, 60]. While to solve the similarity search, extra resources are required. So, it is not feasible to use this brute force method to sequentially scan the whole database to process similarity queries.

Traditionally, to enable fast process data stored in the database, filter-and-refine framework is used [114]. The basic idea is to get the results by a multi-step: In the first step, an easy-to-compute or obvious distance function, which is the lower bound of the actual distance, filters out most objects that have no possibility to be the qualifying results. The candidates returned by the filtering step are then validated by using the original complex similarity measure in the second step. Similarly, to process the operations on the tree-structured data based on similarity measure, distance-embedded lower bounds can also be integrated into this framework to reduce the number of expensive similarity distance computations and speed up the search.

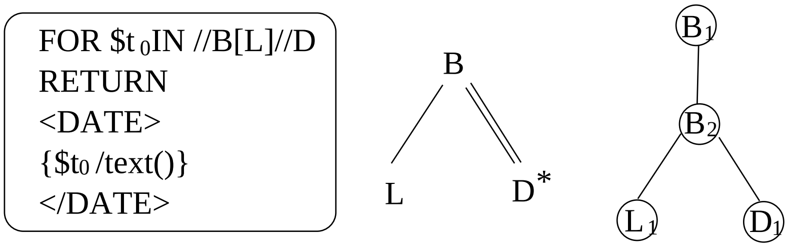
Since the real edit distance is of high computational cost, the efficiency of the multi-step strategy is apparently determined by the efficiency of the filtration step. K. Kailing et al [56] presented a set of filters for structural and content information in trees. However, their filters are for unordered tree models and, at the same time, the structural and content information separately are considered separately in their lower bounds. According to our observation, to design a good filter for rooted ordered labeled trees, the order information between sibling nodes in the tree structure is important for evaluating the distance between trees. Furthermore, the content conveyed by the tag name and the structure of the trees should be explored together to avoid loss of information. Thus, the first purpose of this thesis is to solve the similarity search problem efficiently on XML data by deploying the filter-and-refine framework which is based on a well-defined, easy-to-compute and accurate lower bound distance.

1.5 Motivation for Pattern Query Study

As mentioned above, searching for all occurrences of a twig pattern in the XML database is an core operation in XML query processing. In recent years, many methods ([69, 20, 73, 28, 74, 55]) have been proposed to match XML twig queries efficiently.

In the foremost works ([123, 10]), the query patterns are decomposed into binary structural relationships (either parent-child or ancestor-descendant relationships). Each binary relationship is processed using structure join techniques and the final match results are obtained by “stitching” individual binary join results together. This approach is not optimal due to the uncontrollable intermediate results. Bruno et al. [20] propose a novel holistic approach named TwigStack, which guarantees that each intermediate path solution can contribute to the final solutions for queries which consist entirely of

AD edges. However, when queries contain any PC relationship, TwigStack is non-optimal since it may output a large size of intermediate matches to the individual path expressions which do not contribute to final answers. The recently proposed algorithms, TwigStackList [73] and TJFast [74], proposed by Lu. et al., guarantee the optimality for queries in which PC relationships only occur under the non-branching query nodes and thus slightly enlarge the optimal query class. iTwigJoin proposed in [28] is optimal to AD-predicate-only or PC-predicate-only queries, or 1-branching-node-only queries. However, the optimality for branching query nodes with PC relationships is still an open problem.



```
FOR $t0 IN //B[L]//D
RETURN
<DATE>
{$t0 /text()}
</DATE>
```

(a) (b)

Figure 1.6: Example of Sub-optimal Processing

Another interesting observation is that all the above holistic approaches solve the problem by producing the matching bindings for ***all*** nodes in a twig query. However, in a practical application, this requirement is not necessary. In the XQuery expression, all the matches of certain query nodes are required. However, for other query nodes, only the existence of their matches are required. Query nodes whose matches should ***all*** be returned are referred to as **distinguished** nodes, and those used only for qualifying the structural relationships of a query are referred to as **existential** nodes. For example, in the XQuery shown in Figure 1.6.a, only D is the distinguished node, while B and L are existential nodes. A straightforward approach to answer this query is to postprocess the results of the previous methods and do an appropriate *projection* on the matches of those interesting nodes and remove the redundant query answers which appear in multiple matches. For example, for the twig query in Figure 1.6.a and the data in Figure 1.6.b, all previous algorithms (e.g. TwigStack, TwigStackList, TJFast) output three intermediate path solutions (B_1, D_1),(B_2, D_1) and (B_2, L_1). Through projection and redundancy removal, the real answer D_1 will be retrieved. From the above example, we can see that such a two-steps approach has two problems: (i) it outputs many matching elements of the *existential* nodes that obviously are not required in the original query; and (ii) even if only matching elements for the *distinguished* nodes are considered, prior algorithms still show the non-optimality by outputting many matches of *distinguished* nodes that do not belong to final answers [20, 74, 28]. Therefore, previous approaches output "irrelevant" element matches and "false" element matches.

In this thesis, we analyze the sub-optimality of the prior algorithms, and propose novel efficient holistic twig join methods to process the queries which emphasis the difference between the *distinguished* nodes and the *existential* nodes. Through our work, the optimal query class is essentially enlarged.

1.6 Contribution

The main contributions of this thesis are in two areas: enhancement of the similarity query and the twig pattern query on XML data.

1. The contribution of this thesis on similarity XML query processing can be summarized as follows:

   From the description above, we know that the bottle-neck of solving the XML query problems associated with similarity is the distance measure of XML data. As it is show in Section 1.1, the XML data are usually modeled as labeled tree or graph structures. The generic distance measure is edit-based distance. However, the edit distance function is computed using dynamic programming algorithm and the cost is very high [125, 99, 105, 124]. In this thesis, we propose a new distance measure between XML data. The measure function is based on the transformation of the XML data into its binary tree representation. The structural features and the content information conveyed by the node label can be totally reserved by this transformation. However, the new presentation is propitious to study the effect of edit operations on the tree. The q-gram-like structures on the trees are used in our methods. These miniature structures capture the local pattern of each data. And based on counting the frequency of all these structures, we can get a vector representation for each data: each element in the vector is defined as the number of occurrences of the corresponding miniature structure of the dataset. The vector elements together describe the whole features of the XML tree structure. Thus, each object is transformed to a sparse vector with $|T|$ non-zero items and the original tree edit distance space is transferred to the vector space with L_1 norm distance. The L_1 distance between the vectors is proved to be a close lower bound of the edit distance between the original trees. The intuition here is that more similar the XML data structures are, more common miniature structures they should share.

We also design and analyze novel algorithms which embed the lower bounds into a multi-step framework to solve the similarity search problems. The computation of the distance on the vector is only $O(|T|)$ for each comparison. With this lower bound, most of the computation of the real distance, with time complexity

$$O(|T1||T2|min(depth(T1), leaves(T1))min(depth(T2), leaves(T2)))$$

, can be filtered. Like the q-gram methods which are used to processing similarity search on sequence data, our methods can be generalized according to different dataset characteristics. Through the set of comprehensive performance study, it is shown that our methods are both I/O and CPU efficient.

2. The contribution of this thesis on twig pattern query processing can be summarized as follows:

   Firstly, theoretical analysis of the sub-optimality of previous algorithms is presented. The reason lies in the existence of *matching blocks* on join data streams. There are two kinds of *matching blocks*, i.e. bounded and unbounded matching blocks. Previous algorithm TwigStack [20] suffers the existence of any block including bounded and unbounded matching block. While algorithms TwigStackList [73] and TJFast [74] make progress to efficiently process bounded matching blocks, they still suffer from the existence of the unbounded ones. However, the research in this thesis demonstrates that unbounded matching blocks which involve the *existential* nodes should not result in the non-optimality of holistic algorithms. In addition, an unbounded matching block involving *distinguished* nodes can also be efficiently processed in most cases by selectively caching elements in the main memory.

   Based on the theoretical analysis, two novel algorithms TwigContainment and TwigPrefix using two popular element encoding schemes (i.e. the *containment* and *prefix* encoding schemes) are proposed in this thesis. The new algorithms employ the *bit vector* and *output list* structures (with bounded spaces) to store information and solve the unbounded matching blocks involving *distinguished* nodes. Thus, the new algorithms identify a much larger query class to guarantee the I/O optimality than the existing methods. In addition, it is shown that these two algorithms have the same optimal query class because the theories are developed independent of any specific labeling scheme. Finally, the new algorithms adopt a novel framework for holistic twig pattern matching. Unlike the previous algorithms, which require the postprocessing phrase to do projection on the matches of the distinguished nodes and to remove redundant matching answers, the two new methods proposed in this thesis iterate the input data once and directly output the matching elements of the distinguished nodes.

An extensive set of experimental studies on synthetic and real datasets for performance comparison is presented in this thesis. The results show that TwigContainment and TwigPrefix outperform all tested previous methods. Moreover, although TwigContainment and TwigPrefix have the same optimal query class, the experimental results show that TwigPrefix outperforms TwigContainment in terms of the I/O cost and the total execution time.

1.7 Organization

The rest of this thesis are organized as follows:

- Chapter 2 introduces the background knowledge and related work about XML similarity query and XML pattern query processing.

- Chapter 3 presents the research work on XML similarity query. An efficient method based on the binary tree representation is proposed. Through this method, the XML data tree is transformed into feature-encoded numerical vectors and the distance defined on the numerical vector is utilized to provide pruning power and facilitate the similarity queries on XML data. The experiments show that the pruning power of the new algorithms leads to both CPU and I/O efficient solutions.

- Chapter 4 presents our research work on XML pattern query. The theoretical analysis of the sub-optimality of the previous methods are given. Based on these analysis and the practical requirements of XQuery, two novel algorithms are proposed in this chapter. Experimental results indicate that the new approaches require less memory spaces, while enlarge the optimal query classes.

- Chapter 5 concludes the work in this thesis. This chapter summarizes the main findings of this thesis. At the same time, limitations and future works are also discussed in this chapter.

The work in Chapter 3 is published in [118], and the work in Chapter 4 is based on the technical report of [76].

Chapter 2

Preliminaries and Related Work

In this chapter, I firstly give the background on XML schema languages and the notations utilized in this thesis in Section 2.1 and Section 2.2. Then the background knowledge of XML query processing is introduced which includes the part for XML similarity search and the part for XML pattern query. The review of the research work closely related to this thesis is given as well. The similarity search methods on different types of datasets are briefly introduced in Section 2.3 and 2.3.2. Section 2.3.3 gives the introduction to distance computation on tree-structured data. And various XML similarity measure application is reviewed in Section 2.3.4 . There are lots of research literatures about XML pattern query. According to the processing strategy, they can be classified as relational-based approaches, path navigation approaches and structure join methods. Most of the structure join methods are based on element encoding techniques, and they can be further classified as binary structure join approaches, and holistic twig join approaches. And various indexing schemes have been proposed to facilitate the structure joins. The novel pattern query processing methods proposed in this thesis belongs to holistic twig join methods. Relational-based approaches, path navigation approaches are briefly introduced in Section 2.4.1 and Section 2.4.2. In Section 2.4.3, I present an detailed overview of binary and holistic XML structure join methods. Background information of XML element numbering schemes, which are considered as one of the foundations of structure join, is presented in Section 2.4.3. Review of the indexing techniques designed to facilitate structure join is also given in this section.

2.1 XML Schema

According to the introduction in Chapter 1, we know that XML documents are irregular. However, some XML documents do record related information and share the similar structure. To better describe such XML data structures and constraints, several XML schema languages have been proposed. Now the widely accept schema language is DTD [19], which is a subset of SGML DTD. Essentially, a DTD specifies for every element, the regular expression pattern that the subelement sequences of it need to conform to. The DTD declaration syntax uses commas for sequencing, '$|$' for (exclusive) OR, parenthesis for grouping and the meta-characters, '?', '*', and '+' to denote respectively, zero or one, zero or more and one or more occurrences of the preceding term. The DTD can also be used to specify the attribute for an element (using the $<$!ATTLIST$>$ declaration) and to declare an attribute that refers to another element (via an IDREF field). Figure 2.1 illustrates part of DTD of the XML document shown in Figure 1.1. However, DTD is not required for each document. If a document has a DTD and conforms to it, then the document is valid.

2.2 Notation

In this thesis, XML data are modeled as rooted, ordered, labeled trees. The formal specification of the model for each data is: $T = (N,\ E,\ \Sigma,\ label,\ Root(T))$. N is a finite set of nodes. E is the binary relation on N where each pair $(u,\ v) \in E$ represents the parent-child relationship between two nodes $u, v \in N$. Node u is the parent of node v and v is one of the child nodes of u. This is used to represent the structural information

$<$!ELEMENT MovieDB (Movie | Director | Actor | $\cdots$)*
$<$!ELEMENT Movie (Title, Year, Genres, Director, Cast, $\cdots$) | (#PCDATA)$>$
$<$!ATTLIST Movie
    id CDATA #REQUIRED
    Language CDATA #IMPLIED $>$
$<$!ELEMENT Title (#PCDATA) $>$
$<$!ELEMENT Year (#PCDATA) $>$
$<$!ELEMENT Genres (Genre)$^+$ $>$
$<$!ELEMENT Genre (#PCDATA) $>$
$<$!ELEMENT Director (FirstName, LastName, Movie, $\cdots$) | (#PCDATA) $>$
$<$!ATTLIST Director $director_{id}$ $>$
$<$!ELEMENT Cast (Actor | Actress)$^+$ $>$
$<$!ELEMENT Actor (FirstName, LastName, Movie, $\cdots$) | (#PCDATA) $>$
$<$!ATTLIST Actor $actor_{id}$ $>$
$\cdots$ $\cdots$

Figure 2.1: An Example of XML DTD

between the elements and their subelements, and between elements and their attributes. There exists only one root note, denoted as $Root(T) \in N$ in a data, which has no parent. Every other node v of the tree has exactly one parent ($parent(v)$) and it can be reached through a path of edges from the root. The nodes in the reaching path of v are ancestors of v, denoted as $ance(v)$. Recursively, the nodes reached through v are descendants of v, denoted as $desc(v)$. The nodes which have a common parent v (all the children of u, i.e., $children(v)$) are siblings. The order of the siblings from left to right is significant. Σ is the finite alphabet of tag names and attribute names and $label : N \rightarrow \Sigma$ is a total function. $|T|$ is the number of nodes in tree T, or the size of T.

The $depth$ of a node $v \in N$, denoted as $depth(v)$ is the number of edges on the path from $root(T)$ to v. The out-degree of v, $deg(v)$, is the number of children of v. These definition can be extended such that $depth(T)$ and $deg(T)$ denotes the maximum depth and degree respectively of all the nodes in T. A node without children is a leaf, otherwise an internal/inner node. The number of leaves of T is denoted as $leaves(T)$. Let $T(v)$ be the subtree of T rooted at node $v \in N$. The *preorder traversal* of $T(v)$ is obtained by visiting v and then recursively visiting $T(v_k)$ ($v_k \in children(v)$, $k = 1 \cdots i$) in order. Similarly, the *postorder traversal* of $T(v)$ is obtained by first visiting $T(v_k)$ ($k = 1 \cdots i$) in order, and then v. The *preorder number* and *postorder number*, denoted as $pre(v)$ and $post(v)$ is the number of nodes preceding v in the preorder and postorder traversal of T respectively.

2.3 XML Similarity Search

For many databases, such as multimedia databases, DNA databases, financial databases, medicine databases etc., retrieval of data that are similar to a given reference object is an core operation. Although data can always be scanned sequentially, the amount of disc I/O for the large database make such method prohibitive. Indexing methods are the most primary and direct means to facilitate speedy search.

2.3.1 Traditional Similarity Search Methods

The basic idea is to get the results of similarity query by the multi-step filter-and-refine approach: In the first step, an easy-to-compute or obvious distance function that lower bounds the actual distance is evaluated to filter out the objects that are impossible to be the answer. Then the candidates returned by the filtering step are validated by using the original distance in the refinement step. Indexes are used to prune the searching space and to reduce the amount of data fetched in response to a query and meet the performance requirement. To perform nearest neighbor search, the branch-and-bound searching strategy is the usual choice: The lower bound of the actual distance between the query object and the data indexed are computed using the query object and the corresponding index entry. A pessimistic bound is updated and maintained during the evaluation. The data indexed by the entries which have lower bound exceeding the pessimistic bound can be safely pruned and need not to be fetched from the disc. The data indexed by the remaining entries should be further evaluated to eliminate the false positive.

The lower bound computation should make sure the correctness of the results. So the results are always complete, leading to 100% *recall*. Therefore, the main performance measurement of the indexing methods is *precision*. The less false positives remain, the more effective the index is. That means less data will be fetched from disc to be further evaluated.

The Indexes which support similarity search on numeric multi-dimensional space have been intensively studied [34, 50, 97, 13, 14, 72, 93, 119]. B-tree [34], ISAM indexes, hashing binary trees, are designed for indexing data based on single-dimensional keys, and are not suitable to deal with similarity search which is based on the distance function of multiple parameters. R-tree [50, 97, 13] and its variations are well known to yield good performances for the similarity search on the multi-dimensional points and objects with spatial extents. The basic idea of R-tree and its variations is to hierarchically partition the data space into a manageable number of smaller subspaces. Spatial points and objects are indexed by their associating subspace. However, a poorly designed partitioning strategy may lead to unnecessary multiple path traversal and corrupt the performance of the index. The R-tree-based index deteriorates rapidly when the dimensionality is high. This is because overlap in the directory increases rapidly with increasing dimensionality of data. Many methods have been designed to deal with such “dimensionality curse” problem [14, 72, 93, 119]. Recently, several indexing approaches were proposed to address the similarity search problem on transaction datasets [8, 83, 77]. Extending the common methods from numerical, ordered domains to the transactional data (or marketing data) is not straightforward. The reasons are: (i) Data domains do not have a natural order; (ii) The dimensionality of the transactions is very large, and the datasets are very sparse. Thus these research work partition the search space according to some clustering methods.

2.3.2 Approximate String Matching Problem

The *Approximate string matching* problem is to find the approximate occurrences of a pattern in a data string. This problem usually measures the query pattern and the data with edit distance functions [43, 106]: The substrings of data are signifies, by dynamic programming, for at most k editing operations (insertions, deletions and changes) are needed to convert the substring to the pattern. However computing the edit distance between strings requires time quadratic to the length of the strings in worst case, and therefore, not applicable to large sequence databases.

Q-gram distance of strings is an alternative distance measure in connection with *approximate string matching* problem [102, 47]. Let Σ be a finite alphabet, and let Σ^* be the set consisting all strings over Σ, and Σ^q all string of length q over Σ. The definition of q-gram distance is:

Definition 2.3.1 (q-gram distance between strings). For a string $x_1 = a_1a_2 \cdots a_n$, let $v = a_ia_{i+1} \cdots a_{i+q-1}$, for some i, then x_1 has *occurrence* of v. Let $G(x_1)[v]$ denote the number of the *occurrences* of v in x_1. Then the *q-gram distance* between two string x_1 and x_2 is:

$$D_q(x_1, x_2) = \sum_{v \in \Sigma^q} |G(x_1)[v] - G(x_2)[v]|. \tag{2.1}$$

Example 1. *Given two strings “VACATION” and “VOCATION”, the 3-gram of them are ($\#\#V$, $\#VA$, VAC, ACA, CAT, ATI, TIO, ION, $ON\#$, $N\#\#$) and ($\#\#V$, $\#VO$, VOC, OCA, CAT, ATI, TIO, ION, $ON\#$, $N\#\#$) respectively. Symbol $\#$ is appended to make sure that each character in the strings is in 3 3-grams. Thus, their 3-gram distance equals 6.*

Theorem 2.3.2. *For any $x, y, z \in \Sigma^*$,*

1. $D_q(x, y) \geq 0, D_q(x, x) = 0$*;*

2. $D_q(x, y) = D_q(y, x)$*;*

3. $D_q(x, y) \leq D_q(x, z) + D_q(z, y)$*;*

It is easy to prove the properties of q-gram distance in Theorem 2.3.2. However, q-gram distance is not a metric, since two different strings can have 0 q-gram distance.

To solve the *approximate string matching* problem, processing all the data positions is rather slow. Filtration of data is a widely adopted technique to reduce the string area processed by dynamic programming. One way is to develop necessary conditions for a data area to include an approximate match of the pattern. These conditions often deal with q-grams of the pattern. The intuition is that whenever an approximate match occurs, it has to resemble the original pattern, which is reflected by the existence of the same q-grams in the pattern at the approximate matching position. It has been proved that any edit operation destroys at most q q-grams of the original strings. Thus, q-gram distance can be deduced as a lower bound of the edit distance and can be a filtration on the similarity search. However, as mentioned above, q-gram distance is not an accurate distance measure. So, for the similarity search, it can be used as filtration, but refinement step to eliminate the false positive is required.

2.3.3 Similarity Measure Between Tree-structured Data

Many data mining techniques (for example, nearest neighbor classification methods, cluster analysis, and multidimensional scaling methods) are based on similarity measures between objects. There are essentially two ways to obtain measures of similarity. First, they can be obtained directly from the objects. Alternatively, measures of similarity may be obtained indirectly from the feature vector distance of the objects. Instead of measuring similarity, we can also measure the dissimilarity which is the dual problem of similarity measure. There are many ways to measure the similarity between trees, for instance, the largest common sub-tree and the smallest common super-tree evaluation, the tree edit distance, the alignment and transferable ratio between two trees [9, 86, 49, 125, 126, 105, 124]. Among these measurements, the editing-based distance (*tree edit distance*) is mostly adopted and the focus of this thesis is limited on this measure.

Like the string edit distance measure, all the tree edit distance measures are based on the set of primitive editing operations that can transfer one tree into another. In paper [125], three kinds of operations on ordered labeled trees have been proposed:

- **relabel**: Changing the label of a node v of T.
- **delete**: Deleting a non-root node v means making the children of n become the children of the $parent(v)$ and then removing v (The children are inserted in the place of v as a sequence in the left-to-right order of the $parent(v)$).
- **insert**: Inserting v as a child of v' in T and making v the parent of a consecutive subsequence of the children of v'. Insertion is the complement of deletion.

Let $\lambda \notin \Sigma$ denote a special blank tag name. The cost function $\gamma : (\Sigma \bigcup \{\lambda\}) \times (\Sigma \bigcup \{\lambda\}) \to \mathbb{R}$ is assigned to each edit operation:

$$\begin{aligned}
&\gamma(a \to b), \text{ where } a,\ b \in (\Sigma \textstyle\bigcup \{\lambda\}) \text{ and } a \neq b \\
&\quad a = \lambda,\ \text{means insertion} \\
&\quad b = \lambda,\ \text{means deletion} \\
&\quad otherwise,\ \text{means relabeling}
\end{aligned} \tag{2.2}$$

And this cost function is constrained to be a metric. The generic similarity metric on ordered labeled trees is unit cost edit distance. An *edit script* between T_1 and T_2 is a sequence of edit operations turning T_1 into T_2. The cost of a edit script is the sum of the cost of all the operations. Then $treedist(T_1, T_2)$, the edit distance between T_1 and T_2, is defined as the minimum cost of the edit scripts that transform T_1 into T_2. And the corresponding scripts are the *optimal edit scripts* between T_1 and T_2. (The *optimal edit script* is not unique.)

An *edit operation mapping*, (M, T_1, T_2) (or M without confusion), between the nodes of T_1 and T_2 can be used as the graphic representation of an *edit script* between them. Assuming that there is an ordering between the nodes of trees and that $T_1[i]$ is the ith node of tree T_1 and $T_2[j]$ is the jth node of tree T_2, (i, j) defined in M means $T_1[i]$ should be changed to $T_2[j]$ if $T_1[i] \neq T_2[j]$; or $T_1[i]$ remains unchanged if $T_1[i] = T_2[j]$. If there is no pair defined in M which containing i as the first integer, then ith node in T_1 is deleted. If no pair in M contains j as the second integer, then jth node in T_2 is inserted. The edit operation mapping is one-to-one mapping and preserve the sibling and ancestor relationship between T_1 and T_2. The cost of a mapping can be defined as:

$$\gamma(M) = \sum_{(i,j)\in M} \gamma(T_1[i], T_2[j]) + \sum_{i\in I} \gamma(T_1[i] \rightarrow \lambda) + \sum_{j\in J} \gamma(\lambda \rightarrow T_2[j]) \qquad (2.3)$$

, where I, J are the sets of nodes not touched by M in T_1 and T_2 respectively. It has been proved [125] that for a edit operation script Sc from T_1 to T_2, there exists a mapping M between them that satisfying $\gamma(M) \leq \gamma(Sc)$; and for a mapping M, there is a Sc such that $\gamma(Sc) = \gamma(M)$. So,

$$treedist(T_1, T_2) = \min\{\gamma(M) | M\ is\ a\ mapping\ from\ T_1\ to\ T_2\} \qquad (2.4)$$

Hence, the edit distance computation can be achieved by computing the minimum cost mapping.

Polynomial algorithms exist to compute the tree edit distance and the corresponding edit script. The algorithms are all based on the classic dynamic programming techniques

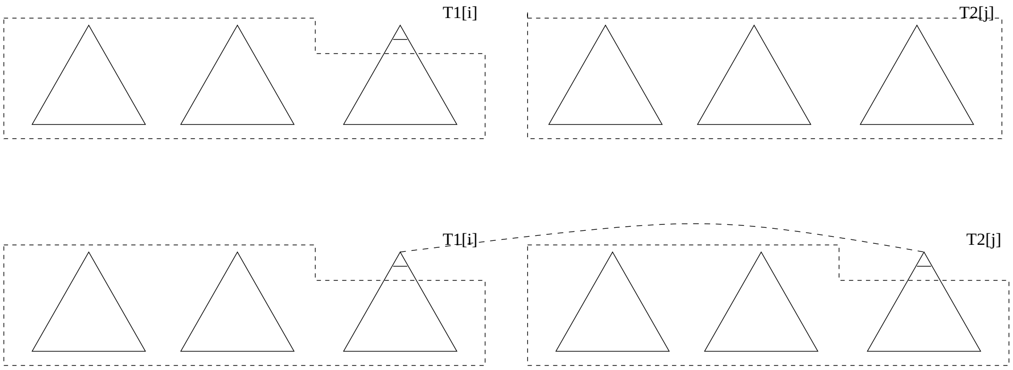


Figure 2.2: Cases of Forest Distance

and most of them are simple combinatorial algorithms. A simple recursion is given for the computation [17]:

Lemma 2.3.3. *Let two forest $T_1[l(i_1)\cdots i]$ and $T_2[l(j_1)\cdots j]$ consist of the nodes $l(i_1)\cdots i$ and the nodes $l(j_1)\cdots j$ from T_1 and T_2 respectively (according to* postorder number*), where $l(v)$ retrieves the leftmost leaf of subtree $T(v)$. Then i and j are the* rightmost *roots (if any). We have,*

$$forestdist(\theta,\theta) = 0$$

$$forestdist(T_1[l(i_1)\cdots i],\theta) = forestdist(T_1[l(i_1)\cdots i-1],\theta) + \gamma(T_1[i] \rightarrow \lambda)$$

$$forestdist(\theta, T_2[l(j_1)\cdots j]) = forestdist(\theta, T_2[l(j_1)\cdots j-1]) + \gamma(\lambda \rightarrow T_2[j])$$

$$forestdist(T_1[l(i_1)\cdots i], T_2[l(j_1)\cdots j])$$

$$= min \begin{cases} forestdist(T_1[l(i_1)\cdots i-1], T_2[l(j_1)\cdots j]) + \gamma(T_1[i] \rightarrow \lambda), \\ forestdist(T_1[l(i_1)\cdots i], T_2[l(j_1)\cdots j-1]) + \gamma(\lambda \rightarrow T_2[j]), \\ forestdist(T_1[l(i_1)\cdots l(i)-1], T_2[l(j_1)\cdots l(j)-1]) \\ +forestdist(T_1[l(i)\cdots i-1], T_2[l(j)\cdots j-1]) + \gamma(T_1[i] \rightarrow T_2[j]). \end{cases}$$

Proof. (This proof is given in [125].) The first three equations are trivially true. To prove the last equation, consider a minimum cost mapping M between $T_1[l(i_1)\cdots i]$ and $T_2[l(j_1)\cdots j]$ shown in Figure 2.2.

Case 1: i is not touched by a mapping line (The first case in Figure 2.2). Then $(T_1[i] \rightarrow \lambda) \in M$ and the first case of equation 4 is applies.

Case 2: j is not touched by a line. Then $(\lambda \rightarrow T_2[j]) \in M$ and the second case of equation 4 applies.

Case 3: i and j are both touched by lines (The second case in Figure 2.2). This implies that $(i, j) \in M$. Otherwise, let $(i, h), (k, j) \in M$. If i is to the right of k (or is the proper ancestor of k), then h should be to the right of j (or be the proper ancestor of j). Both are impossible since j is the right most root.

Since the edit operation mapping reserves the ancestor descendant relationship, any node in subtree $T_1[i]$ can only touched by nodes in $T_2[j]$. Hence,

$$\begin{aligned}
&forestdist(T_1[l(i_1) \cdots i], T_2[l(j_1) \cdots j]) = \\
&forestdist(T_1[l(i_1) \cdots l(i) - 1], T_2[l(j_1) \cdots l(j) - 1]) \\
&+forestdist(T_1[l(i) \cdots i - 1], T_2[l(j) \cdots j - 1]) + \gamma(T_1[i] \rightarrow T_2[j]).
\end{aligned}$$

The third case of equation 4 follows.

□

Lemma 2.3.3 suggests a dynamic program. The value of $forestdist(\, , \,)$ depends on a constant number of subproblems of smaller size. Hence, the time complexity is bounded by the number of subproblems of $T_1[l(i_1) \cdots i]$ times the number of subproblems of $T_2[l(j_1) \cdots j]$. The number of the subproblem is quadratic to the size of the forests respectively.

The work in [125, 60] proved that the subproblem size can be reduced by revising the recursion definition. Zhang et.al rewrite the last equation of Lemma 2.3.3 and have the following lemma:

Lemma 2.3.4. *Let $i_1 \in anc(i)$, $j_1 \in anc(j)$. We can have:*

(1) If $l(i) = l(i_1)$, and $l(j) = l(j_1)$, then

$$\begin{aligned} & forestdist(T_1[l(i_1)\cdots i], T_2[l(j_1)\cdots j]) \\ & = treedist(T_1(i), T_2(j)) \\ & = min \begin{cases} forestdist(T_1[l(i_1)\cdots i-1], T_2[l(j_1)\cdots j]) + \gamma(T_1[i] \to \Lambda), \\ forestdist(T_1[l(i_1)\cdots i], T_2[l(j_1)\cdots j-1]) + \gamma(\Lambda \to T_2[j]), \\ forestdist(T_1[l(i_1)\cdots i-1], T_2[l(j_1)\cdots j-1]) \\ +\gamma(T_1[i] \to T_2[j]). \end{cases} \end{aligned}$$

(2) If $l(i) \neq l(i_1)$, and $l(j) \neq l(j_1)$, then

$$\begin{aligned} & forestdist(T_1[l(i_1)\cdots i], T_2[l(j_1)\cdots j]) \\ & = min \begin{cases} forestdist(T_1[l(i_1)\cdots i-1], T_2[l(j_1)\cdots j]) + \gamma(T_1[i] \to \Lambda), \\ forestdist(T_1[l(i_1)\cdots i], T_2[l(j_1)\cdots j-1]) + \gamma(\Lambda \to T_2[j]), \\ forestdist(T_1[l(i_1)\cdots l(i)-1], T_2[l(j_1)\cdots l(j)-1]) + treedist(T_1(i), T_2(j)). \end{cases} \end{aligned}$$

Lemma 2.3.4 makes sure that before the computation of $treedist(T_1(i), T_2(j))$, all distances $treedist(T_1[i_1], T_2[j_1])$ are available if i_1 (or j_1) is in the subtree of $T_1(i)$ ($T_2(j)$) but not in the path from $l(i)$ ($l(j)$) to i (j). After the computation of $treedist(T_1(i), T_2(j))$, all distances $treedist(T_1(i_1), T_2(j_1))$ are available, where $l(i_1) = l(i)$ and $l(j_1) = l(j)$. The $keyroots$ of T is defined as follows in [125].

$$keyroots(T) = \{root(T)\} \bigcup \{u \in N(T) \mid v \text{ has a left sibling } \}$$

The *special* subforest $F(v)$ of T is the forest under node $v \in keyroots(T)$. For a node $v \in N(T)$, the ***collapsed depth*** of v, $cdepth(v)$, is defined as the number of $keyroot$ ancestors of v. Also $cdepth(T)$ is the maximum collapsed depth of all nodes in T.

Lemma 2.3.5. *For an ordered tree T, the relevant subproblem size w.r.t. the keyroots is bounded by $O(|T|cdepth(T))$. And $cdepth(T) \leq \min\{depth(T), leaves(T)\}$.*

Thus, the algorithm proposed in [125] to compute edit distance between trees is of $O(|T_1| \times |T_2| \times \min(\textit{depth}(T_1), \textit{leaves}(T_1)) \times \min(\textit{depth}(T_2), \textit{leaves}(T_2)))$ time complexity.

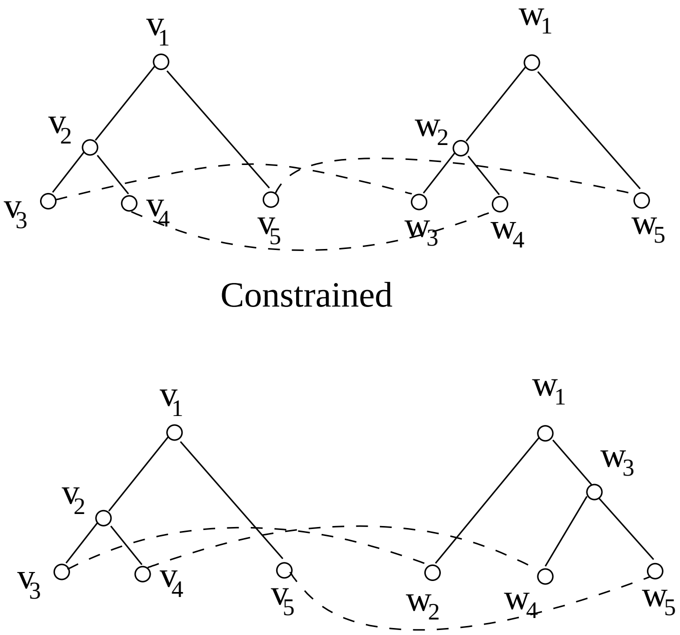
In paper [60], the worst case time complexity of the edit distance computation is reduced further by decomposing a tree T into disjoint paths, *heavy paths*. First the nodes of T is classified as *heavy* or *light* as follows: The root is *light*. The child node of the internal nodes with the maximum size is classified as *heavy*. The edge to the *light* nodes are *light edges*, while the one to the *heavy* nodes are *heavy edges*. The *light depth* of node v, $ldepth(v)$, is the number of light edges on the path from $root(T)$ to v. In the paper, Klein proved that the number of *relevant* subproblems w.r.t. the light nodes is bounded by $O(|T|ldepth(T))$ and for any $v \in N(T)$, $ldepth(v) \leq log|T| + O(1)$. Thus, the worst case time complexity is bounded to $O(|T_1|^2|T_2|log|T_2|)$

The main difference between various tree-distance algorithm is the set of allowing edit operations. The earlier work in [96] allows insertion and deletion of single nodes only at the leaves and relabeling of nodes anywhere in the tree. Definition in [125, 99, 105, 124] allow insertion and deletion of single nodes anywhere in a tree. In [124] a new distance metric based on a restriction of the mappings between two trees is proposed. The intuition is that two separate sub-trees of T_1 should be mapped to two separate subtrees in T_2. The demonstration of constrained mapping is shown in Figure 2.3. The constrained edit mapping is a kind of restricted mapping which satisfies:

1. $1 \leq i \leq |T_1|, 1 \leq j \leq |T_2|$;

2. the mapping is the one to one mapping, preserving sibling order and ancestor order

3. For any triple (i_1, j_1), (i_2, j_2) and (i_3, j_3) in the mapping, let $lca()$ represent least



Constrained

Not Constrained

Figure 2.3: Examples of Constrained Mapping

common ancestor function, $t_1[lca(i_1, i_2)]$ is a proper ancestor of $t_1[i_3]$ iff $t_2[lca(j_1, j_2)]$ is a proper ancestor of $t_2[j_3]$

While, the alignment distance in [105] allows only the insertion before the deletion. In an alignment $\mathcal{A}$ of two trees T_1 and T_2, the nodes labeled with λ (*space*) are inserted into T_1 and T_2 to obtain two new trees T_1' and T_2' with the same structure. And then the nodes on T_1' are paired with the corresponding nodes on T_2': pair (a, b) means replacing if $a \neq b$, (a, λ) means deletion operation and (λ, b) means insertion. A score are assigned for each pair. The *value* of $\mathcal{A}$ is the sum of scores of all pairs of it. Note that a standard assumption is that the score scheme γ satisfies triangle inequality. And the optimal alignment is one that minimize the value of all possible alignments. The *alignment distance* is the value of the optimal alignment. Figure 2.4 is an example of alignment.

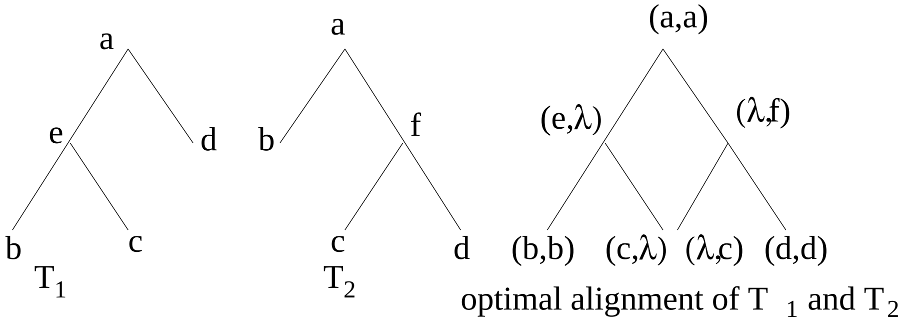


Figure 2.4: Alignment of Tree T_1 and T_2

2.3.4 XML Applications Associating Similarity Measure

Just as mentioned previously, an XML data is formally modeled as a rooted ordered labeled tree. So most literatures use the similarity measure between trees to solve the problem of XML data. Guha et al. [48] presented an approximate XML join based on the tree edit distance. In their method, XML documents are transformed into their corresponding preorder and postorder traversal sequences. Then the maximum of the string edit distance of the two sequences is used as the lower bound of the tree-edit distance. They also proposed to use a constrained tree-edit distance, which is of complexity $O(|T_1||T_2|)$, as the upper bound of the generic tree edit distance to reduce the computation further. In addition, they use the reference sets to take advantage of the fact that the tree edit distance is a metric, thus reducing the actual amount of edit-distance computations between pairs of trees. However, the complexity of computing the proposed lower bounds is still $O(|T_1||T_2|)$ (i.e., the complexity of sequence edit distance computation), and it is not scalable to large dataset.

In the recent work, Kailing et.al. [56] presented a set of filters grounded on structure and content-based information in trees. They proposed using the vectors of the height histogram, the degree histogram and the label histogram to represent the structure as well as content information of trees. The lower bound of the unordered-tree edit distance can be derived from the L_1 distance among the vectors. They also suggested a way to combine filtration to facilitate similarity query processing. However, their filters are for unordered trees and cannot explore the structure information implicitly depicted by the order of siblings. Moreover, their lower bounds are obtained by considering structure and content information depicted by tag names separately. In Chapter 3, we suggest combining the two sources of information to provide accurate lower bounds for the tree-edit distance. And we compare the performance of our algorithm against the histogram filtration methods.

Garofalakis and Kuma [44] correlate streams of XML data through approximate matching in small space. They presented an efficient approximation of the tree edit distance by embedding the tree-edit distance metrics (allowing a *move* operation in addition to the basic operations) into a numeric vector space with L_1 distance norm. In their method, XML trees are hierarchically parsed into valid subtrees in different phases. Then the multi-set of valid subtrees is obtained by parsing the tree. The vector representation is defined as the characteristic vector of the multi-set. The L_1 distance of the vectors guarantees an upper bound of distance distortion between two trees. However, the method fails to give a constant lower bound on the tree-edit distance to facilitate the retrieval of exact answers to the similarity queries based on similarity measure.

pq-Grams was introduced by Augsten et al. [12] as approximation of tree edit distance for ordered trees. pq-gram anchored at a node u in the tree consists of $p-1$ ancestors and q children of u. The missing components are made up by appending nodes with tag $*$. Accordingly, the pq-gram profile of a tree T is a vector consisting of the occurrences of all the pq-grams in T and the pq-gram distance of trees is the distance of the corresponding pq-gram profiles. The distance thus defined is sensitive to the inner node changes and weight local changes less than distributed changes. The effectiveness of this orientation depends on the application.

In change detection scenarios, two versions of the same document are given and the difference is computed. Cobèna [33] takes advantage of existing element IDs, which cannot be assumed for joins of data from different sources. Chawathe et al. [25] present a heuristic solution for unordered trees that runs in $O(n^3)$ time and for many cases in $O(n^2)$. The X-Diff algorithm by Wang et al [113] allows leaf and sub-tree insertion and deletion and node relabeling. To achieve $O(n^2 \times deg(T) \log(deg(T))$ runtime, they match only nodes with the same path to the root node. The distance measures presented above are evaluated between pairs of documents.

Weis and Naumann [115] proposed a similarity measure between XML documents in a duplicate detection framework. In the worst case, all pairs of elements have to be compared. Puhlmann et al. [91] improved the efficiency by applying the Sorted Neighborhood method to nested objects. Both approaches assume a known, common schema of the matched documents and require a configuration step.

2.4 XML Pattern Query

To answer pattern queries on XML data, it is not efficient to measure the similarity between the query patten and the data directly. Firstly, the information about the position in the document tree where a pattern matching can occur is not available in advance. Secondly, it is difficult to define the similarity measure between query pattern and data since XML pattern query consists of path expressions containing wildcards and regular expressions. According to the searching strategy, previous XML pattern query methods can be classified as relational-based pattern query methods, path navigation-based pattern query methods, structure join-based methods. There also exists some methods which are based on query transformation instead of query decomposition. In this section, we systematically study all of these methods.

2.4.1 Relational-based Pattern Query Processing

In practice, XML data can be managed by traditional database, such as relational or object-oriented database. Relational database implementation is regarded as a practical approach because of its wide deployment in commercial world and its mature RDBMS technologies, e.g., indexing, concurrency control and transaction management. Some previous work processes XML pattern query by using RDBMS [98, 38, 41]. They mainly solve the following three subproblems [98]:

(1) Physical schema design: transferring the arbitrarily nested XML schema into the flat table schema of relational database. The recursive structure of the XML data requires special processing.

(2) Query mapping: converting XML queries to corresponding SQL queries over the tables obtained from transformation.

(3) Result construction: exporting the existing data as XML

The first subproblem is a tradeoff between the storage cost and query processing performance. This depends on the features of the data (the shape, the size and the recursive property etc). The naive approach is to transform each element into a relation, with each attribute of the element as one column of the table. The relationship between elements is implemented by foreign keys. However, there is no one-to-one correspondence between the attributes of XML elements and the columns of relational tables. Furthermore, this causes the fragmentation problem: To be space optimal, the irregularity of XML requires to store different elements in different tables. However, this transformation may cause too many join operations on multiple tables for XML query processing. If multiple elements are mapped to a single table, there may be much waste on storage space.

One type of transformation is on generic XML data without schema assumption. The methods proposed in [38] employ a heuristic to achieve efficient relational schema design. The frequently occurring portions of XML documents are stored in a relational system, while the remainder is stored in an overflow graph. The intuition is that the "interesting node groups" usually are the frequent ones. Then less joins are required for many queries. The authors of [41] classified the transformation methods into 6 categories: According to the structural mapping, they proposed *Edge* table, *Binary* table and *Universal* table. And according to the value storage, there can be *value inlined* and *value outlined* strategies. Edge strategy completely fragments the input document into one table with schema $(source, childNo, tag, target)$. This strategy incurs many (self) joins over a large table to answer even simple queries. Furthermore, redundant information is stored since tags are repeated. At the same time, updating operation is costly. Binary strategy clusters the edges according to tags and horizontally partitions the Edge table. Then joins are performed over much smaller tables and better performance is achieved for query evaluation. Tags are not redundantly stored any more. Universal table stores all edges in a single universal table. It is obtained by outer join all the Binary tables and stores each node-to-leaf path in a tuple. The query performance can be improved by Universal table by reducing the join operation. However, there still exits too much redundancy in this table.

Shanmugasundaram [98] demonstrated how to map the XML schema into relational schema by utilizing the DTD specification to evaluate powerful queries over XML documents. The shared inline techniques is proposed to inline as many subelements as possible in the element tables. If an element is of a shared type (the in-degree of it in DTD graph is larger than 1), or it is recursively defined, or it consists of set of subelements, then it cannot be inlined. Instead, separate table is constructed for it. However, the tables for shared elements may lead to extra joins to answer path expressions. Hybrid inline techniques try to solve this problem by inline some shared elements, i.e., the elements with in-degree larger than 1 which are neither recursively defined nor consisting of set subelements. However hybrid inline method may incur more SQL sub-queries. Obviously, it is a fundamental tradeoff between reducing number of queries and reducing number of joins for each query. In addition to XML schema, the relational schema works at different efficiency according to different workloads. In [18], the authors proposed to optimize the schema transformation by exploring the space of possible transformations under the guidance of the cost evaluation which is defined according to the XML schema, the data statistics and the query workload. However, the set of possible configurations is very large - possibly infinite. Thus, the greedy algorithm is used to select efficient mapping alternatives for a variety of workloads. The selected configurations are robust to variations on workloads and superior to the all-inlined strategy. However, the efficiency of this methods depends on the accuracy of the statistics derivation.

To convert semistructured queries on XML to SQL, the path expressions need to be transformed. In [98], the authors gave a framework. Firstly, the relation corresponding to the context of the root path expressions need to be identified, and be transformed to FROM clause of SQL. Then, joins between tables are required if the elements are not in the same table. The recursive path expressions can be transformed to the union of two SQL fragments within a least fix-point operator. Arbitrary and complicated queries need to be transformed into simple (recursive) path expression first, and then to SQL queries separately.

Relational implementation show limitations on converting the results of SQL queries to complex structured XML documents, since the construction may contain tag variables and grouping operations and complex elements. In [98], the authors proposed some solutions to these problems. However, these require the processing outside relational engineer, which abandoned the mature optimization techniques of RDBMS.

2.4.2 Path Navigation-based Pattern Query Processing

XML query languages (e.g. XPath and XQuery etc) specify the path expressions which can be answered by navigating the irregular structures of data. However, such query processing may be very inefficient due to the navigation of the whole data graph, especially when the objects are scattered on different locations of the disk. Structural summaries or indexes of XML database can speed up query evaluation by restricting the search to only the relevant portion of the XML data. Thus the extraction of indexes based on structural summary of XML data has received a lot of research attention [46, 80, 58, 32]. Some of them are based on relational-based implementation.

The indexes for the semistructured data can be adopted to process XML queries [46, 36]. In [46], *DataGuide* is defined as the concise summaries on the semistructured data. It describes every unique label path of a source exactly once and encodes no label path that does not appear in the source and each object in *DataGuide* can have a link to its corresponding target set in the source. Hence, we can find all source objects reachable via a label path in time proportional to the length of it. One source database may have multiple *DataGuide* among which the optimal one should be explored. Furthermore, multiple label paths can reach the same object and undistinguishable in *DataGuide*. To solve these problems, *strong DataGuide* is proposed [5]. It ensures that the set of all label paths sharing the same target set with some path l in the source data equals to the set of label paths in the *strong DataGuide* that share the same target set with l. Thus, it can induce a one-to-one correspondence between source target sets and the *DataGuide* objects. T-index [80] indexes all sequences of objects connected by a sequence of path expressions defined by a template. 1-index indexes all objects reachable through an arbitrary path expression from the root: Two nodes are equivalent (same entry in index) if the set of paths into them from the root is the same. It is a non-deterministic version of the strong data guide. 2-index indexes all pairs of objects connected by an arbitrary path expression. In T-index, objects that are indistinguishable w.r.t to a class of paths defined by a path template are grouped into one equivalence class. Fine equivalence classes can be constructed efficiently by using bi-simulation. *DataGuide*s and 1-Index suffers two problems. Firstly, they are inefficient when processing queries starting with descendant predicate steps and queries containing wildcard "*". Secondly, they do not support the branching queries.

APEX [32], $F\ \&\ B$-Index [58] and *Index Fabric* [36] construct the index on refined paths or pre-defined query patterns, instead of storing all paths from root to leaves. *Index Fabric* extends $DataGuide$ for text search. It keeps all label paths starting from the root and encodes each label path with data value as a string, which can be efficiently indexed by patricia trie. And the queries on keywords for elements are processed as string search. In [58] the structural summary of schema-less data are constructed by using the notion of inverse edges which capture the information about both in-coming and out-going paths. This is so called Forward and Backward-Index ($F\ \&\ B$-Index). It has been improved that the $F\ \&\ B$-Index is the smallest index graph that covers all branching path expressions over graph data. Unfortunately, $F\ \&\ B$-Index is usually too big to be loaded in the main memory. When the database is huge, $F\ \&\ B$-Index is almost the same as the original data. To solve this problem, the index definition scheme need to find the optimal tradeoffs between the size of the index and the queries to be covered. *APEX* [32] takes advantages of query workload to mine the frequent query path expressions and summarizes data paths that appear frequently in query workload. In addition, it also maintains all paths of length two. So, *APEX* is flexible and faster than *strong DataGuide*.

2.4.3 Structure Join-based Pattern Query Processing

To process XML queries with recursive predicates, i.e., the AD relationship predicates, the previously mentioned top-down evaluation can be inefficient - the whole subtree rooted at an element needs to be tested. On the contrary, structure join methods utilize certain element numbering scheme which encodes the position information of the elements, to verify the structural predicates on elements [123, 22, 101, 116, 107, 88, 74]. Based on this, various approaches of binary structure join [41, 79, 10, 104, 103, 98, 123] and holistic twig join [20, 28, 74, 55] were proposed. The former class of approaches firstly process the binary relationship constrains which are obtained by decomposing the tree-pattern queries, and then merge-join the intermediate partial results to get the final answers. While, the holistic twig join methods try to answer the queries as a whole.

Element Numbering Schemes

The main purpose of numbering/encoding XML elements, denoted as function $num(\ )$ on element is to allow fast identification of relationships between elements. (In some literatures, the encoding positional numbers are also called as labels. However, the label are specifically used as the node names of trees in this thesis.) There are two classes of popular numbering schemes in the literatures, i.e., the *containment numbering* (or *range/region numbering*) [35, 123] and *prefix numbering* schemes [107, 88, 74]. The *containment numbering* scheme supports efficient evaluation on AD and PC structural relationships between elements. But this kind of schemes is not capable of supporting data updates. In *prefix numbering* schemes, the number of an element is decided by the number of its parent and its own tag name. Therefore, it can support the structural relationship verification by string matching methods. Meanwhile, it deals with data update more flexibly than the *containment numbering* scheme. Recently, many researchers have begun to design dynamic XML labeling schemes to handle data updates [22, 101, 116].

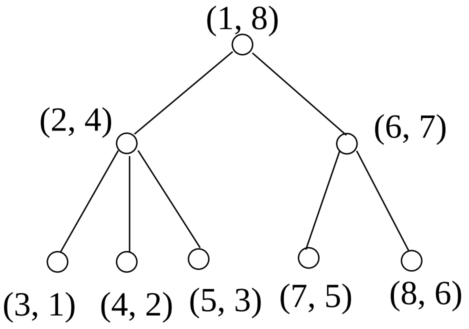


Figure 2.5: Dietz's Numbering Scheme

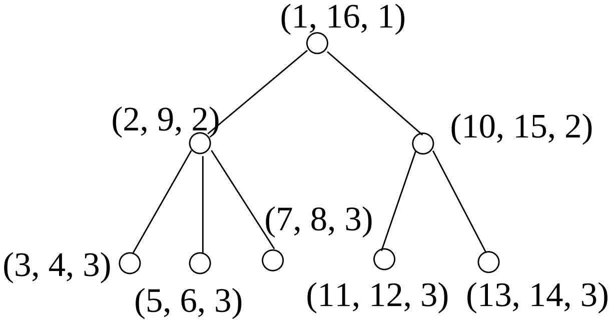


Figure 2.6: Containment Numbering Scheme

Two earlier numbering schemes designed to decide the document structure is the Dietz's [39] scheme and Lee's scheme [65]. Dietz's encode each node in the tree by its preorder and postorder numbers. As we all know, the preorder number of a descendant is larger than that of its ancestor, while the postorder number of a descendant is smaller than that of its ancestor. Thus $pre(u) < pre(v)$ and $post(u) > post(v)$ is the conditions to identify the AD relationship, which can be evaluated in constant time. An example of Dietz's encode is shown in Figure 2.5. Lee's scheme models the documents as *complete k-ary tree*, where k is the largest fanout of the tree. Each node is encoded by the breadth-first traversal number of the enlarged tree. Then equation $num(parent(u)) = \lfloor (num(u) - 2)/k \rfloor + 1$ can be used to determine PC relationship. Obviously, the space overhead of this scheme can be prohibitively high. At the same time, the updates of the documents cannot be processed straightforwardly by these two methods.

The first containment encoding is ascribed to the work of Consens and Milo [35], who discussed a fragment of PAT text searching operators for indexing text database. Then Zhang et al. [123] introduced it to XML query processing using inverted list. Each inverted list records the occurrences of an element type. Each occurrence e is indexed by its document number, its position and its nesting depth within the document, denoted by $num(e) = (docID,\ LeftPos : RightPos,\ level)$. $LeftPos$ (or $RightPos$) is

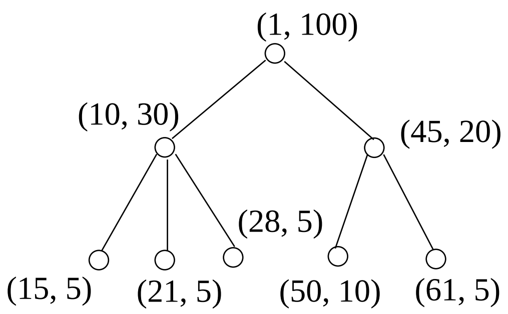


Figure 2.7: Example of Interval Numbering Scheme

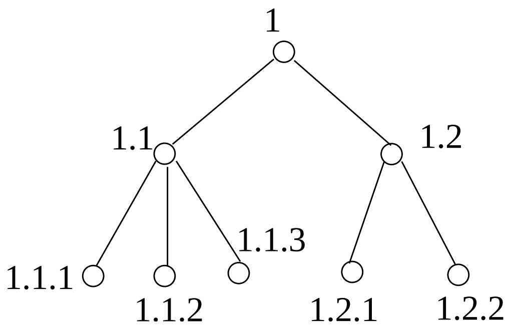


Figure 2.8: Example of *Dewey ID* Scheme

the position number of the start (or end) tag of the indexed element. The numbers are sequentially arranged during depth-first traversal. Figure 2.6 shows a example of the numbering scheme. Thereby, the position range of the ancestor elements should contain that of the descendants, and the parent node level equals to the children level minus 1. Such scheme is widely adopted in [52, 54, 20, 28]. However, the update processing based on containment numbering is costly: The insertion of a new node leads to re-labeling of all the ancestor nodes and all the nodes following it.

Interval encoding [69] is a variation of containment encoding, which aims to alleviate the update awkward processing. Each element u is identified by a pair of numbers $num(u) = (order,\ size)$. For a node u which is the parent of v: $order(u) < order(v)$, and $order(v) + size(v) \leq order(u) + size(u)$. For two sibling nodes u' and v', if u' is the predecessor of v' in preorder traversal, then $order(u') + size(u') < order(v')$. The interval encoding of the above example is shown in Figure 2.7. Obviously, extra space can be reserved to accommodate future insertions. However, the scheme will collapse if no extra space is available.

To our best knowledge, *Dewey ID* numbering scheme is the first prefix numbering scheme. It comes from the work of Tatarinov et al. [107] to represent XML order in the relational data model. The *Dewey ID* labeled each element as follows: (1) The root element is numbered by one-character string “1”; (2) The non-root elements are encoded as the concatenation of their parent's numbers and their positions among the siblings. Thus, the ancestor number of an element can be derived directly from its own. For example, in Figure 2.8, if a label of an element is "1.1.2", then it has 2 ancestor and the labels of them are "1", "1.1" respectively. This encoding scheme supports efficient evaluation of structural relationship between elements by prefix checking of the numbers. However, from the *Dewey ID* of an element alone, we cannot derive the tag name of its ancestors.

*Extended Dewey*s [74] incorporates not only the structural relationships, but also the element name information into the encoding. From the *extended Deway* number of an element alone, the names of all the elements in the path from the root to it can be derived. The rational is to encode the element name under a specific parent context by using the modulo function: For a element e, all its possible child element names are ordered as $< t_0, t_1, \cdots, t_n >$. If the child element e' of e has tag name t_i then a integer x is assigned to e' such that $x \; mod \; n = i$. For text values, $x = -1$. Similar to *Dewey ID*, the number of e' in *extended Dewey* are the concatenation of e and x assign to it. The sibling information can also be encoded. Specifically, given an element e_i with tag $t_{i'}$ and its left sibling element e_j (if exists) with tag $t_{j'}$ and number y, the *extended Dewey* number of e_i, $num(e_i)$, is $num(parent(e_i)).x$, where x is computed as follows:

$$x = \begin{cases} i' & e_i \text{ is the left most child of } e_p; \\ \lfloor \frac{y}{n} \times n \rfloor + i' & \text{otherwise, if } i < j; \\ \lceil \frac{y}{n} \times n \rceil + i' & \text{otherwise.} \end{cases}$$

According to the number of an element, the tag names of the elements from the root to it can be decoded by a *finite state transducer* (FST). The symbols of the FST are non-negative integers and -1; The states are the tag names and an additional state, named PCDATA; For a state t, if its ordered child element tags are $< t_0, t_1, \cdots, t_n >$, then the transition function is defined as $\delta(t, x) = t_k$, where $k = x\ mod\ n$. The output is the current state after transition. Figure 2.9 is part of the transducer constructed according to the DTD definition shown in Figure 2.1. For clarity, the tag names are represented by the capital letters and the PCDATA state is omitted in Figure 2.9. Then XML path pattern matching can be directly processed by string matching. For example, through FST, we element labeled as "$2.0.1$" is associated with path "MovieDB.Movie.Cast.Actor" in the data, then its straightforward to identify that it matches a path pattern "//Cast/Actor". In the worst case, the space complexity of the FST is quadratic to the size of the tag name alphabet and time complexity is linear to the length of the path, but independent of the complexity of the schema definition.

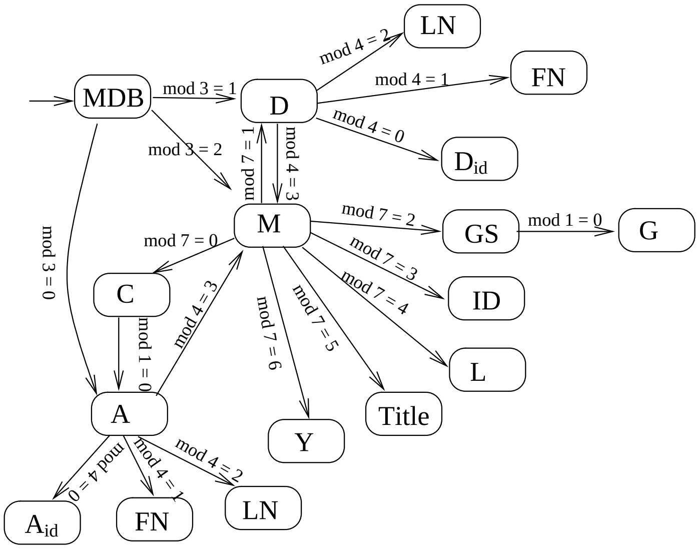


Figure 2.9: The Transducer of the Extended Dewey Labeling Scheme

O'Neil et al. [88] introduced a variation of prefix labeling scheme called *ORDPATH*. Unlike the *extended Dewey* [74], the main goal of *ORDPATH* is to gracefully handle insertion of XML nodes in the database. It uses odd numbers at the initial document encoding. When there is an insertion on the document, the even number between two odd numbers catenated with another odd number is labeled on the new node. Although the insertion is processed in linear time, *ORDPATH* wastes half of the numbering space by using odd numbers initially. This numbering scheme lose the level information too. At the same time, the even number seeking process is time consuming if the insertions follows multiple deletion. Wu et al. proposed in [116] a prime numbering scheme. This scheme assign to each node a prime number. The position encoding of a node is the product of its parent's number and its own number. Thus, prime numbering scheme can be viewed as an extension of the prefix labeling. Then for two nodes u and v in the tree, u is an ancestor of v iff $num(v)$ MOD $num(u) = 0$. This scheme can be used to encode the dynamic ordered XML tree as follows: *Simultaneous Congruence* values of *Chinese Remainder Theorem* can determine the orders among siblings. When the document is updated, it only requires to recalculate *Simultaneous Congruence*. However, the recalculation is much time consuming. The CDBS (*Compact Dynamic Binary String*) scheme presented in [66] is orthogonal to specific labeling schemes. The order is maintained by the lexicographical orders of the binary strings and the elaborately designed binary string insertion methods. By using CDBS, the re-labeling is totally unnecessary. However, if the insertion always occurs at the same place, the size of the numbers will increase fast.

Binary Structure Join Methods

Some of the previous work [41, 79, 10, 104, 103, 123, 98] has typically decomposed the twig pattern into a set of binary relationship between pairs of query nodes, i.e., the PC relationships and the AD relationships. Then the twig query can be processed by two steps: Firstly, evaluate each of the binary structural relationships against the XML database and a set of element pairs which satisfy the binary relationship predicate is generated. Secondly, "stitch" together the basic matches obtained by the first step to get the final results. For example, a pattern query expression shown in Figure 2.10.(a) can be decomposed into the binary structural predicates shown in Figure 2.10.(b).

For most of the structure join methods, the data structure referred to as element

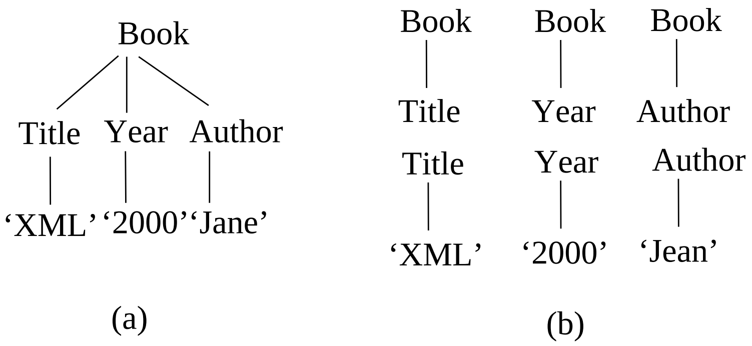


Figure 2.10: An example of Twig Query Decomposition

streams is used to store the inverted lists of the encoding numbers of elements of the same type. The encoding numbers in each stream are sorted in ascending document order. When processing the queries, the element stream S_q satisfying the node predicate of q, which is under consideration, is retrieved from the disk and iterated by an associated one-way cursor in the sorted order. The element (actually, the encoding number of the element) pointed by the cursor is referred to as cursor element. An example of the streams is shown in Figure 2.12. To evaluate the structure join matches is actually to join the elements from two streams which satisfy the structural predicates. The structure specification can be efficiently checked by the numbering techniques mentioned in Section 2.4.3. The related work mentioned in this section are all based on *containment numbering* scheme.

The *Multi-predicate Merge Join* (MPMGJN) proposed in [123] is essentially a form of nested-loop join. This approach scans the same element streams multiple times in case the XML data is nested. The scanning of the parent query node elements consists of the outer loop, while the scanning of the elements of the child query node consists of the inner loop. However, for each outer loop, the scanning of child query node stream need not to be start from the first element stored in it. The relationship between two elements from the streams of two query nodes q and q' (q' is the ancestor query node of q) are given in Figure 2.11. Assume that e_q is the first element in the stream of q which satisfies

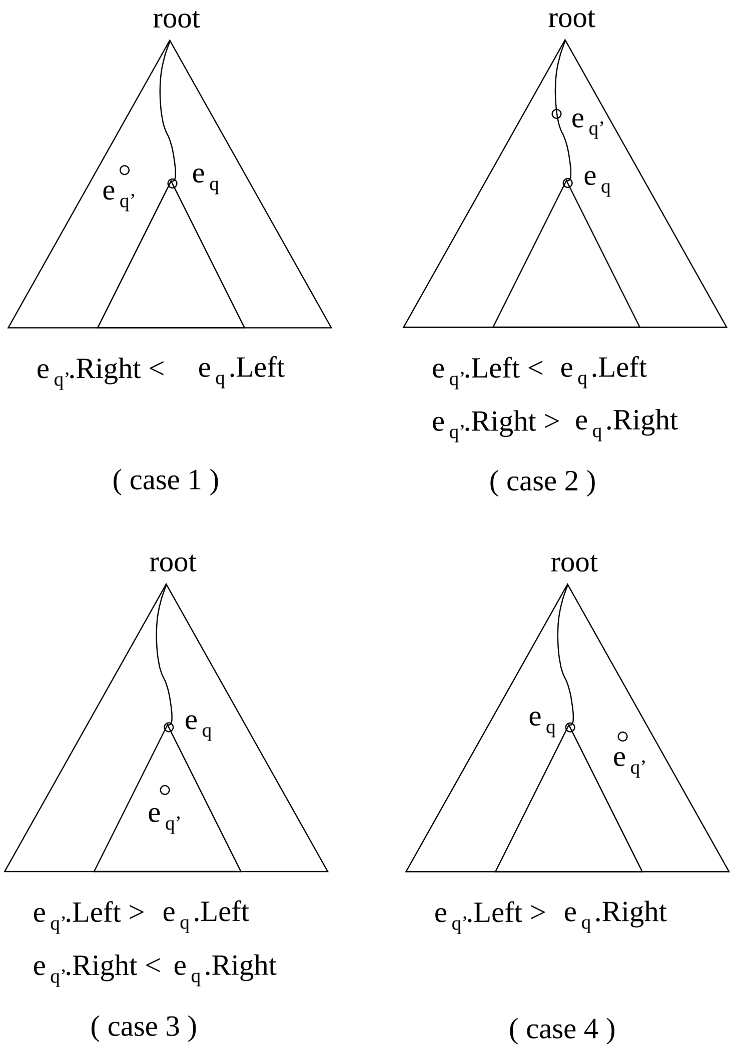


Figure 2.11: Relationship Cases for Two Elements e_q and $e_{q'}$

the relationship predicate with the previously accessed element $e_{q'}$ in stream of q'. Then e_q and $e_{q'}$ should be of relationship shown in Case 2 of Figure 2.11. For any element $e'_{q'}$ following $e_{q'}$ in $S_{q'}$, its child or descendant elements cannot precede e_q in S_q. Thus, the inner loop for $e'_{q'}$ can start correctly from e_q, instead of the head of stream q. The authors of [69] differentiated 5 types of subexpressions for the path expression decomposition: the one with unit components (the single element or single attribute), the one with two element relationship specification, the one with element and attribute specification, the one with *Kleene closure* (symbolized by ‘*’ and ‘+’) specification and the one with union specification. Accordingly, the author gave 3 types of join methods. The nested loop can be safely avoided in EA-join since there is no recursive definition on attributes. EE-Join solves the join between elements like MPMGJN, but seeking is not done on data records directly. Instead, the algorithm searches the B^+-tree indexes (XISS) for element and attribute names, values and structures. The interval numbering scheme is utilized to determine ancestor-descendant relationship in constant time. However, the elements in child query node streams still need to be scanned for multiple times.

To avoid the multiple scan of the element streams, the *Stack-Tree* algorithm [10] utilizes an internal stack to store a subset of the elements from stream of q . The elements in one stack from the bottom to the top are nested in one path of the data. Thus, For the Case 1 in Figure 2.11, cursor element e_q cause $e_{q'}$ to be popped out from the stack because it cannot contribute to the future matching results. While for Case 2, $e_{q'}$ is already iterated and pushed into the stack before e_q is reached. It remains in the stack after e_q is pushed into the stack to encode the matches of the binary relationship constrain. After output the required results, the stream of q can be safely advanced. If there exists no such element and the stack is empty, e_q can be safely skipped. Hence, only one sequential scan of q's stream is necessary.

In the second step of binary structure join, the results evaluated in the previous step need to be “stitched” together. The method based on selectivity and intermediate result size estimation is required [79, 7, 71, 90, 68] to decide the optimal join order. The details are not included in this thesis, since these topics are not closely related to the work here.

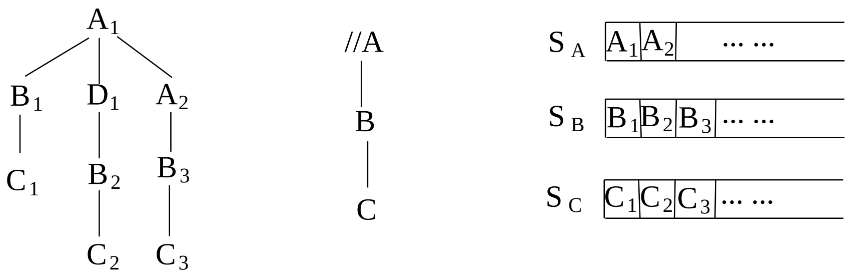
Although all the above methods were proposed to improve the efficiency of binary structure join, there exists a basic limitation of these decomposition-based methods. They may output large number of intermediate results which do not contribute to the total answers to the query path pattern, not to mention the answers to the twig pattern query. For example, if the query $//A/B/C$ is proposed on the data shown in Figure 2.12.

Firstly, we need to compute matches to the PC predicate: A/B or B/C. 2 answers will be retrieved for A/B query, i.e., (A_1, B_1) and (A_2, B_3). And 3 answers will be retrieved for B/C query, i.e., (B_1, C_1), (B_2, C_2) and (B_3, C_3). However, (B_2, C_2) cannot contribute to the total answer of the query. So, if any binary structural predicate is of low selectivity, the input size and time expense for the later join will be quite high.

Holistic Twig Join Methods

In order to solve the problem of large amount of intermediate results, a series of holistic twig query methods have been proposed to process the twig pattern as a whole [20, 28, 74, 55]. In these methods, the elements are also stored in the encoding number streams and only the streams satisfy the query predicates of each query node are retrieved. Meanwhile, each query node q is associated with a stack SK_q, in which each item consists of a pair: the positional encoding number of an element retrieved from the stream S_q, and an pointer points to an item in $S_{parent(q)}$ (TJFast [74] uses different data structures as mentioned later). The stack is used to encode the partial/total answers to the twig pattern query. And the elements stored in it satisfy two requirements: (1) When the element is pushed into the stack, the algorithms make sure that it and the top element of $S_{parent(q)}$ satisfy the containment relationship. After the element is pushed into its stack, a pointer is built to associate it to the top element of $S_{parent(q)}$. (2) The AD relationship between elements in the same stack are implicitly encoded, i.e., elements are strictly nested from bottom to top.

In paper [20], Bruno el at. proposed a novel path matching algorithm, called PathStack to process linear path expressions. In this method, the twig query pattern is decomposed into multiple root-to-leaf path patterns. The entire path queries are processed in a top-down query predicate checking style. Before an element e_q is pushed into its stack S_q, all elements in the other stacks which end before it should be popped out first to



(a) Data (b) Query (c) Streams

Figure 2.12: An Example of Data, Query and Stream Structures

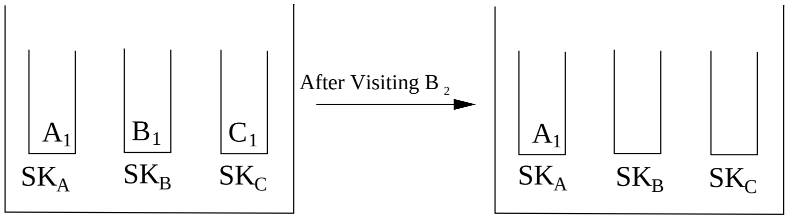
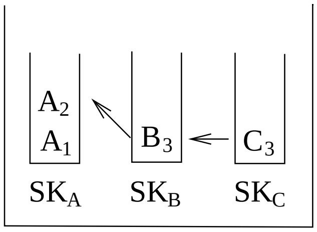


Figure 2.13: Example of Stack Pushing



Output $A_1B_1C_1$

(a) Stack encoding

A_1	B_1	C_1
A_2	B_3	C_3

(b) Query results

Figure 2.14: Stack-encoded Results for Path Query

make sure that all the stacks encodes compactly partial/complete answers of a path query at any time. e_q can be pushed into the stack iff (1) the $S_{parent(q)}$ is not empty and (2) e_q

and the top element of $S_{parent(q)}$ satisfy the relationship constrain between $parent(q)$ and q. Once an element is pushed into the leaf query node stack, there must be some answers to the corresponding query path. For the data, the stream structures and path query shown in Figure 2.12.a, 2.12.b and Figure 2.12.c, Figure 2.13 shows the stack operation. In Figure 2.13, A_1, B_1 and C_1 are pushed into the stacks since they satisfy PC relationship and the 2 requirements mentioned above. When element B_2 is iterated, element B_1 and C_1 are popped out since they end before B_2. Figure 2.14.a shows the partial results output and encoded in the stack. And Figure 2.14.b is the results of the path query on the data. Unlike *Stack-Tree*, element B_2 will not be pushed into stacks by PathStack because there is no matching elements in SK_A. When C_2 is iterated, SK_B is empty. So C_2 cannot be pushed into its stack as well. The efficiency of this method lies in two aspect: The stacks deployed can represent in linear space a potentially exponential number (to the size of the query nodes) of answers. Meanwhile, it reduces the query processing cost since only the top element in the parent stack needs to be check each time. The worst-case CPU time cost to solve path queries is linear to the sum of the input streams and the output lists, which is independent of the size of any intermediate binary join results.

Although PathStack method can process the path pattern query as a whole, as for twig patterns, it cannot totally solve the problem incurred by decomposition. Some printed partial answers to the path queries may not be merge-joinable at the branching query nodes. The methods dealing with the twig query as a whole are needed. Holistic algorithm also consists of two steps: In step one, the partial answers of the path pattern queries are output. In step two, the partial answers are joined to get the full answers of the twig pattern. However, in step one, holistic algorithms try to output only the partial answers which is merge-joinable with at least one solution to each of the other root-to-leaf paths. We say that q has an solution extension if there is a solution for the sub-query rooted at q. If the solution consists of only the cursor elements, then q has a minimal match, otherwise, it has possible match. Holistic algorithms retrieve the highest query node q that has a possible match each time. This makes sure that the elements which match the ancestor query nodes of q and can contribute to a total query answer have already been pushed into the corresponding stacks. Thus, after clear the ancestor stack by e_q, if the $S_{parent(q)}$ is empty, the current element cannot be an answer to the query. Otherwise, it is pushed into the stack. Once the stack of a leaf query node is pushed into a element, there should be an partial solution which is merge-joinable with at least one solution to each of the other query paths. Under the holistic twig join scheme, the cursor elements of the streams can be classified into 3 types: the matching elements, the useless elements and the blocked elements. A matching element e_q is in a minimal match of q, but not in any future match of $parent(q)$. Holistic methods can tell a matching element and safely push it in the stack. Useless elements are those which do not participate in any possible match to its query node. It is safe to skip them. The rest are blocked elements. For example, if we propose a query shown in Figure 2.15 on data in Figure 2.17, According to PathStack, the partial answers (A_1, B_1, C_1) and (A_1, B_3, D_1), (A_3, B_2, D_1) and (A_1, B_4, D_2) to the path pattern query $A//B//C$ and $A//B//D$ respectively will be output. Obviously, they cannot be merged into a whole answer. If holistic join methods are used, elements A_3, A_4, B_3, B_4, B_5 are skipped because they are useless. C_1 cannot be pushed into the stack because S_B is empty when it is iterated. Thus, the above output partial solutions are not output by holistic methods.

However, the elements available are the cursor elements and those at the top of each stack. From these information, only minimal matches can be exactly identified. Whether a possible match exists for a node cannot be tell exactly. There are three cases for the blocked elements. In the first case, they are in possible but not in any minimal match

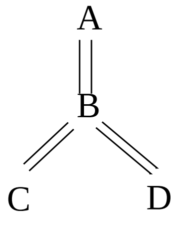


Figure 2.15: Twig Pattern Query (a)

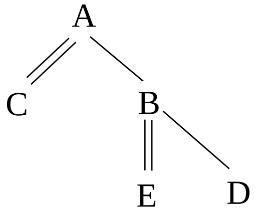


Figure 2.16: Twig Pattern Query (b)

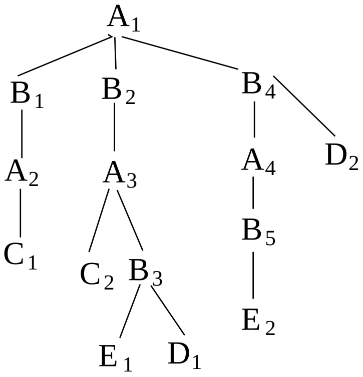


Figure 2.17: The running example of XML data for holistic twig join methods

to the query node. In the second case, they are in a minimal match to its node but is only in the possible match of the parent query node. Or in the third case, they are not in possible match of the parent query node. Thus, advancing any streams of block elements without storing the cursor elements may cause the loss of results. The foremost holistic method, TwigStack [20], relaxed the PC constrains to AD constrains for the internal nodes when it verifies the solution extension to avoid the false dismissal. However, this leads to useless intermediate results. For the query in Figure 2.16 and the data in Figure 2.17, the cursor elements are A_1, B_2, C_1, D_1, E_1, B_2 and D_1 is not in a match to the PC relationship between B and D. However, B_2 contains D_1 and B_2 may be in a match with elements following D_1. Similarly, D_1 may be in a match with elements following B_2. TwigStack pushes A_1, B_2, C_1, D_1, E_1 into the stacks according to the containment relationship although (A_1, B_2, E_1) and (A_1, B_2, D_1) are not the satisfactory partial answers.

The concept of optimal twig pattern matching algorithm is officially defined in [28]. A twig pattern matching algorithm is optimal if it can satisfy the following tree conditions:

1. Every element stream retrieved for the pattern (i.e., whose tag appears in the twig pattern) is scanned only once.

2. None of the intermediate partial solutions output is redundant.

3. The space required by the algorithm is bounded by a factor which is independent of source document size.

For an twig pattern matching method to be optimal, the case that all current elements are blocked should never occur. Apparently, TwigStack is only optimal for path patterns and AD only twig patterns, but sub-optimal to twig query containing PC relationships [20, 74]. The later research work tries to minimize the blocked elements for queries and expand the types of queries which can be optimally answered [73, 28, 74].

Lu et al. proposed TwigStackList in [73]. It makes sure that if there is a PC relationship below the branching node q and its child q_c, the cursor element e_q can be pushed into stack only if it or the elements following it in S_q satisfy the PC predicate with the cursor elements of solution extension rooted q_c. Thus even if there exists PC relationship predicate under branching nodes, the TwigStackList is superior to TwigStack in that it output less useless intermediate solutions. However, for the above example, the output of TwigStackList is the same as that of TwigStack. iTwigJoin [28] increases "parallelism" to access elements with the same tag by the additional "context" information. It uses refined streaming scheme, and partition the streams of elements by $Tag + Level$ context or the more refined prefix-path context. For example, the streams for the data in Figure 2.17 are shown in Figure 2.18. iTwigJoin associates the useful streams according to the context. For instance, the useful B streams of S_A^1 and S_A^3 are S_B^2 and S_B^4 are respectively. The logic iTwigJoin is similar to that of TwigStack, but adopted to process the refined streaming scheme. For the above example, the element B_1 and B_2 will be skipped since the useful stream of node D for $S_B^2 = S_{AB}$ begins with D_2. So these two elements have no descendant extensions in this stream. Thus the intermediate results (A_1, B_2, E_1), (A_1, B_3, E_1), (A_1, B_2, D_1) and (A_1, B_3, D_1) are not output. However, if the element E_2 does not exist in the data, the redundant result (A_1, C_1) and (A_1, B_4, D_2) will still be output. The methods are proved to be optimal only for AD-relationship-only query, PC-relationship-only query and 1-branching-node query.

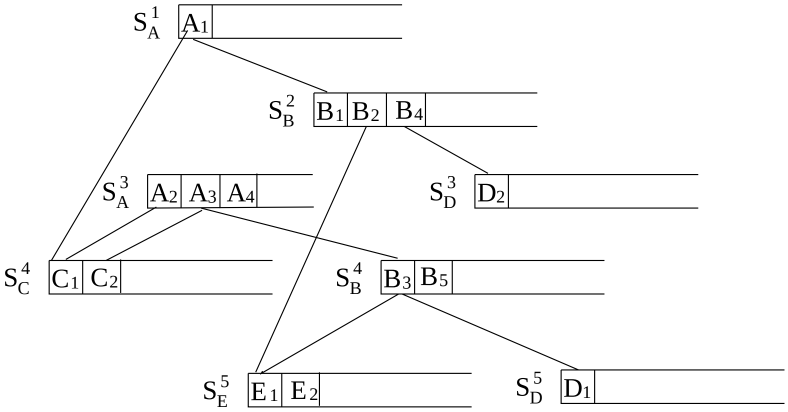


Figure 2.18: The Refined Streaming Scheme of $iTwigJoin$

TJFast proposed in [74] is based on *extended Deway* numbering scheme introduced in Section 2.4.3. From the definition of the *extended Deway* number of an element e, the names of the all the elements in the path from the root to e can be derived directly. Thus, whether elements are satisfying the path pattern queries can be checked by string matching algorithm and only the numbers of elements matching the leaf query node need to be scanned. This fact leads to two benefits: Firstly, the I/O cost is much smaller than the previous methods. Secondly, TJFast can efficiently process path queries contain AD relationships or wildcards “$*$” by string-matching with *don’t care* symbols. Therefore, to evaluate a twig pattern, the only key issue is to determine whether a path solution can contribute to the solutions for the whole twig, i.e., whether it and solutions of other path queries have common element which can match to the branching node. TJFast guarantees that each output partial solution shares common elements from the branching node streams (which are not physically retrieved from the disk) with at least one partial solutions to all the other path queries. The PC relationship on non-branching nodes can be guaranteed by string matching algorithm directly. So when there are only AD relationships under branching nodes, TJFast is proved to be optimal. However, for the data and query shown in Figure 2.17 and Figure 2.16 respectively, the redundant intermediate solution (A_1, B_2, E_1) will still be output.

Structure Join based on Indexed Documents

The previously mentioned structural join methods may still incur unnecessary I/O costs since they need to scan the entire streams, especially in the case where only a small portion of nodes in the streams satisfy the containment relationship. The potential benefits of skipping elements that do not participate in the final twig match by using available indexes are explored in the methods reviewed in this section [31, 54, 20]. They are for both the binary structure join methods and the holistic twig join methods. For example, assume that q' is a query node and q is one of its child node. Let $e_{q'}$ and e_q are the cursor elements of $S_{q'}$ and S_q. If they are in relationship shown in Case 3 and Case 4 of Figure 2.11 and $SK_{q'}$ is empty after pop the elements ends before e_q, then e_q is impossible have corresponding match of node q. Thus, cursor of S_q can be advanced till the first element whose start point is larger than that of $e_{q'}$; If $e_{q'}$ and e_q are in Case 1, cursor of $S_{q'}$ should forward to make the cursor element and e_q are in Case 2, if possible. Otherwise, in Case 3 or Case 4; If $e_{q'}$ and e_q are in Case 2 of Figure 2.11, the elements following $e_{q'}$ in $S_{q'}$ but start before e_q should all be retrieved to push into the stack. These are achieved by indexing the encoding number of each element $(DocId,\ LeftPos : RightPos,\ LevelNum)$.

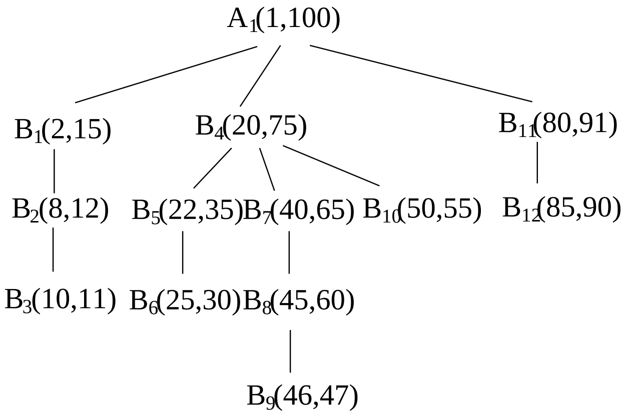


Figure 2.19: An Example of Indexed XML Tree

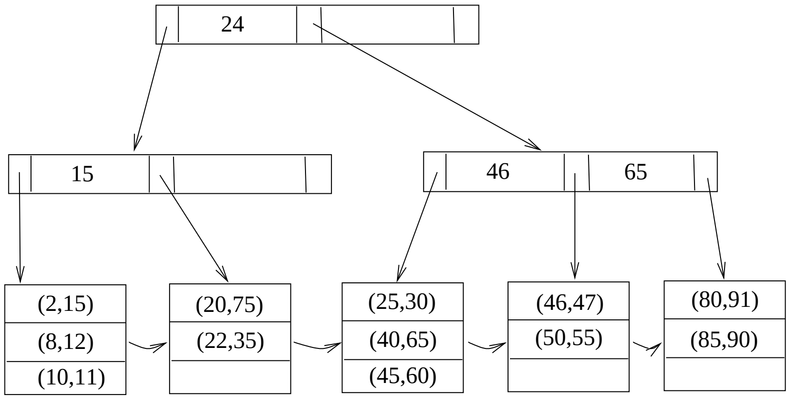


Figure 2.20: B^{+}-tree Indexed

The *Anc-Des-*B^{+} method of [31] builds B^{+}-tree index for the elements in each streams. The indexing key is the $LeftPos$ of the encoding number. The index of element B for the XML data in Figure 2.19 are shown in Figure 2.20. XR-tree (XML Region Tree) proposed in [54] is essentially a B+tree index on the $LeftPos$ of the containment encoding numbers. Figure 2.21 is an example of XR-tree index. In addition, each internal node associates a stab list. A element e with encoding number $(e.start, e.end)$ is called to be stabbed by a key k_i if $e.start \leq k_i \leq e.end$. The elements are stored in the leaf nodes as well as the stab list of the top-most internal node containing a key which stabs it. In [20], TwigStack is extended to TwigStackXB by XB-tree index. XB-tree is like a one dimension R-tree index on the containment encodes and the intervals are arranged according to $(DocId,\ LeftPos)$ as those in B-tree. Figure 2.22 shows the XB-index for the data in Figure 2.19.

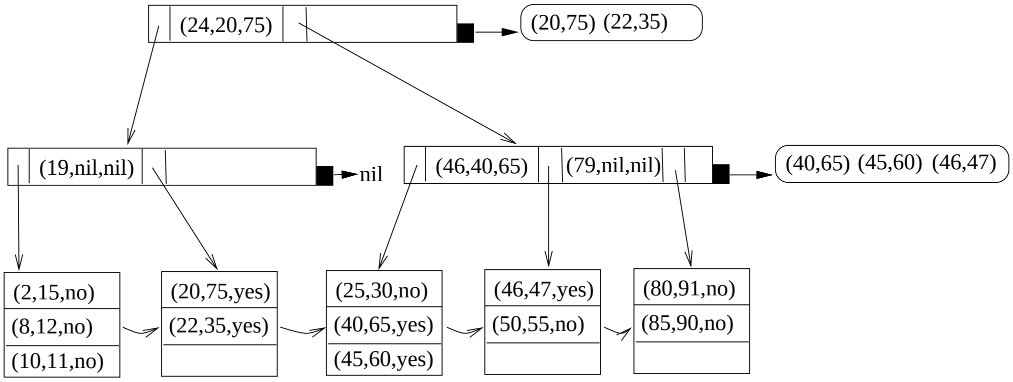


Figure 2.21: XR-tree Index

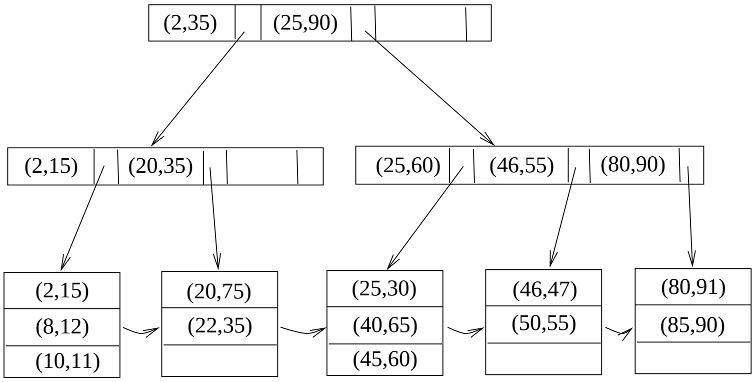


Figure 2.22: XB-tree Index

Jiang et al. [55] proposed TSGeneric$^+$, an novel holistic twig join based on the indexing scheme of the *containment encoding* of elements (e.g., B^+-tree, XB-tree and XR-tree [31, 20, 67] etc.). It is proved that, in addition to the relationship between two query node, the relationship between a query node q and its descendant can be utilized to skip more elements. If the stack of q is empty, even if its descendant nodes have solution extension, there cannot exist a match to the whole twig query. Rather, it is safe to move cursors of the descendants forward to locate a solution extension for q. The authors also gave three heuristics on the order of picking the broken edge in the subtree rooted at q to improve the performance.

2.4.4 Query Processing Method Without Decomposition

The authors of [112, 92] developed methods which solve twig pattern queries as a whole. Both methods transform the XML data trees and queries into sequences and the nodes of the data sequences are stored with position numbers encoding the its positions in the virtual trie indexes. The virtual index structure reduces the amount of data that need to be searched. *ViST* transforms XML data trees and the twig queries into structure-encoded sequences which consist of $(e.label, e.prefix)$ pairs in document order, where $label$ is the tag name of the element e in the XML document tree or the label of the query node, and $prefix$ represents the label path from the root to e. *ViST* performs subsequence matching on the Transformed sequences to find twig patterns in XML documents. One imminent weakness of *ViST* is that the worst-case space requirement of the virtual index structure is high because the prefix of the elements are required to encode the structure. At the same time, the query processing strategy may result in false alarms because the subsequence matching method cannot distinguish the structures in which the two elements are siblings from the structures where two elements have the same prefix. In order to conquer this problem, the authors proposed *constrained subsequence matching* in their later work [111]. In [111], Wang et al also discussed the optimal sequencing strategy with regard to the time and space complexity to index and query XML data, which should be guided by XML schema and data distribution of the dataset. The transformation method of *PRIX* [92] is based on $Prüfer$ encoding of the tree structure which constructs a one-to-one correspondence between a labeled tree and the transformed sequence. Non-matches are filtered out by subsequence search on the indexed sequences. The twig matches are then found by applying refinement. The author proved that the connectedness and the structure verification by the gap and frequency consistency is necessary and sufficient to verify a partial twig match to the structure query. Thus no postprocessing is required.

2.4.5 Query Processing with More Complicate Predicates

Recently, several research work focus to process the complicated query predicates in addition to the PC and AD constrains [53, 75, 120]. They deal with the queries with OR-predicate, ordering predicates and NOT-predicate respectively. The work presented in this thesis can be extended to solve these complicated predicates too.

2.5 Summary

In this chapter, XML schema languages and the formal notations of the XML data model are given first. Then we reviewed the techniques for XML similarity query processing and XML pattern query processing. The review shows that the studies on similarity query processing is not sufficient although this query is the basis for many data manipulations on XML. The efficiency needs to be improved if datasets consist of large sized XML documents. For the XML pattern queries, although intensive research has been conducted previously, the optimal processing of PC relationship constrains is still an open problem. And the optimal query classes need to be enlarged further.

Chapter 3

Similarity Evaluation on XML Data

3.1 Introduction

In this chapter, the study of the structure similarity measure and similarity search on large XML data in huge datasets is presented. These problems form the core operation for many data analysis tasks (e.g., approximate join, clustering, k-NN classification, data cleansing, data integration etc). It is also useful for document management including XML data searching under the presence of spelling errors, version management for XML documents, etc. In practice, similarity query itself is the main data manipulation for multimedia and time-series databases, biological and scientific databases. Since XML is the de facto standard for data exchange on the web, more and more commercial data and scientific data are conveyed in XML documents. Thus, efficiently processing similarity evaluation on XML data poses interesting challenges for database researchers. However, little research have been done on this area. There is still no efficient similarity search algorithm for XML.

The main reason is that data model of XML is different from those of conventional databases. As mentioned in Chapter 1, the data are often with no schema specification. Even if there is schema, the data conforms to it flexibly. Elements and attributes can be optional and elements can occur multiple times. The traditional distance measurements, thus, cannot be used straightforward in this area. Furthermore, in the XML document, the semantics specified implicitly by the relationship between its components. Then the structures play important role on differentiating data. The measurement of XML data similarity can be precise only if this information is exploited and introduced into the measure function. However, this cannot be done directly by the traditional metric.

Now that there are lots of literatures discussing about the similarity measure of the value content, in this chapter, we particularly focus on the that conveyed by the the tree structures and tag names. Usually, the XML data are modeled as rooted ordered, labeled tree-structural data (details are in Chapter 1). The generic distance measure is *edit-based* distance [84]. However, the *tree edit distance* function is computed using dynamic programming algorithm and the cost is very high [125, 99, 105, 124]. Data manipulations based on the tree edit distance directly can be very expensive both in terms of CPU cost and disc I/Os, rendering it impractical for huge datasets.

In this chapter, a structure transformation on rooted, ordered, labeled trees is utilized to develop a novel distance function based on both the structural and the content information. It is proved that the proposed distance function is a lower bound of the tree edit distance. The idea is similar to using a set of q-grams to bound the edit distance of strings and thus filter out dissimilar strings [110]. Given a string S, a q-gram is a contiguous substring of S of length q. If S_1 and S_2 are within edit distance k, S_1 and S_2 must share at least $max(|S_1|, |S_2|) - (k-1)q - 1$ common q-grams. Similarly a tree can be characterized by a set of q-level binary branches, and it is shown that two trees T_1 and T_2 are within edit distance k precisely when they share $[4 * (q-1) + 1] * k$ q-level binary branches. Furthermore, just as string edit distance can be tightened if the positions of the q-grams in the string are also taken into account [102, 47], so too tree-edit distance can be tightened by using information detailing the positions of q-level binary branches in the trees.

By employing the distance function as the lower bound of the edit distance in the filter-and-refine framework, the evaluation of the similarity queries can be solved in two steps: In the filtering step, the lower bound is used to filter out most objects which are not possible to be in the result. The remaining objects are candidates which are validated by the original complex similarity measure during the refinement step. This strategy greatly reduces the number of expensive distance computations in the original space.

The rest of this chapter is organized as follows: Section 3.2 presents the definition of the transformed vector space and the new distance based on it, together with the formal proof of the lower bound theorem. Section 3.3 discusses how to embed the new distance function as the lower bound of edit distance into the framework for similarity search, while in Section 3.4 a thorough experimental study of the new algorithms is presented. Finally, Section 3.5 concludes this chapter.

3.2 Tree Structure Transformation

The key element of the new algorithm is to transform rooted, ordered, labeled trees to a numeric multi-dimensional vector space equipped with the norm L_1 distance. The mapping of a tree T to its numeric vector ensures that the features of the vector representation retain the structural information of the original tree. Furthermore, the tree-edit distance can be lower bounded by the L_1 distance of the corresponding vectors. The lower bound distance evaluation is computationally much less expensive than that of $EDist(T,\ T')$. In this section, the transformation methods and the proof of the lower bound theorem are presented.

3.2.1 Binary Tree Representation of Forests (or Trees)

The proposed mapping of tree structures into a numeric vector space is based on the binary tree representation of rooted ordered labeled trees. For completeness, firstly the binary tree representation of forests (or trees) is briefly introduced. The formal definition of the binary tree is cited from [61]:

Definition 3.2.1 (Binary Tree). A *binary tree* consists of a finite set of nodes. It is:

1. an empty set. Or

2. a structure constructed by a root node, the left subtree and the right subtree of the root. Both subtrees are binary trees, too.

In a binary tree, the edges between parents and the left child nodes are different from those between parents and the right child nodes. We use $T_B = (N,\ E_l,\ E_r,\ Root(T))$ to represent a binary tree. $\forall u,\ v_1,\ v_2 \in N$, if v_1 (v_2 resp.) is the left (right resp.) child of u, then $\langle u,\ v_1\rangle_l \in E_l$ ($\langle u,\ v_2\rangle_r \in E_r$ resp.). A full binary tree is a binary tree in which each node has exactly zero or two children.

There is a natural correspondence between forests and binary trees. The standard algorithm to transform a forest (or a tree) to its corresponding binary tree is through the left-child, right-sibling representation of the forest (tree): (i) Link all the siblings in the tree with edges. (ii) Delete all the edges between each node and its children in the tree except those edges which connect it with its first child. Note that the transformation does not change the labels of vertices in the tree. T_1 and T_2 of Figure 3.1 can be transformed into T_1' and T_2' shown in Figure 3.2. By rotating it, we can get the binary trees $B(T_1)$ and $B(T_2)$ respectively shown in Figure 3.3.[1] The binary tree representation is denoted as $B(T) = (N,\ E_l,\ E_r,\ Root(T),\ label)$ in this chapter.

[1] The appended nodes with label ε and the numbering of the nodes are explained in sections 3.2.3 and 3.3.2, respectively.

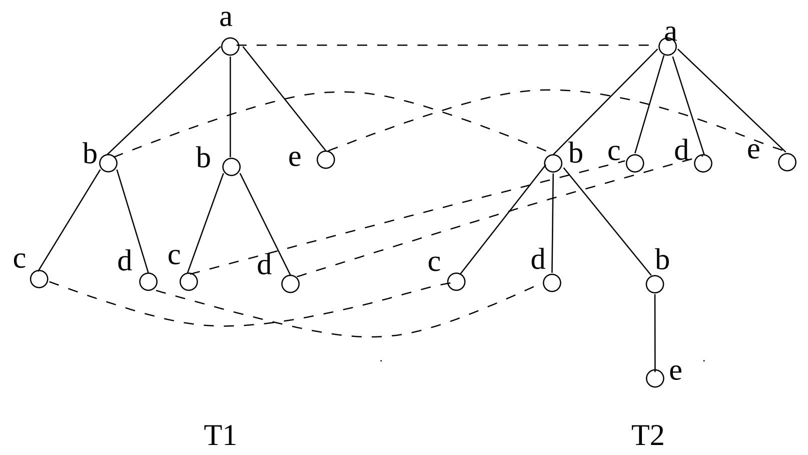


Figure 3.1: Tree Examples

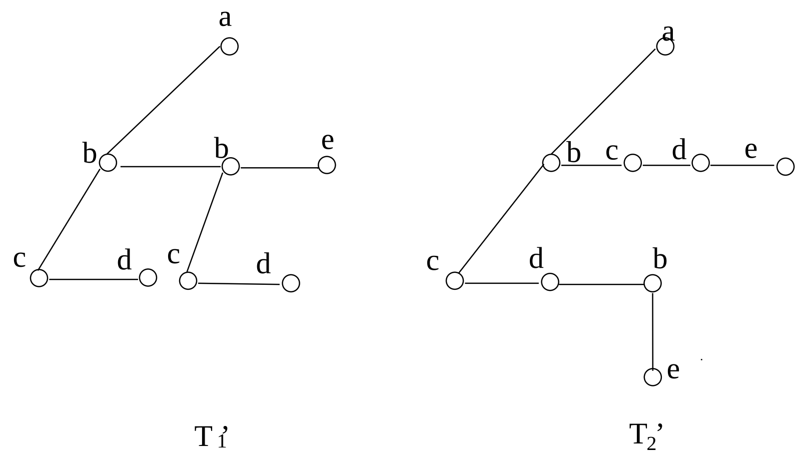
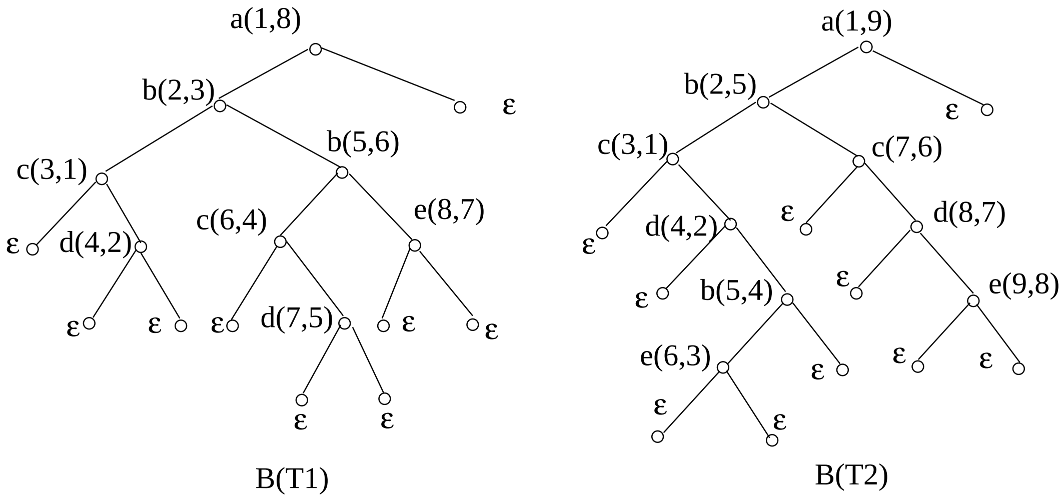


Figure 3.2: Tree Transformation

3.2.2 Observation

The inspiring observation is that edit operations change at most a fixed number of sibling relationships. This is because each node in a tree can have a varying number of child nodes but at most two immediate siblings. This is illustrated in the example of Figure 3.1. The deletion of node b in T_1 incurs five changes in parent-child relationships: It destroys the $(a, b),\ (b, c),\ (b, d)$ edges, while generating the $(a, c),\ (a, d)$ edges. At the same time, this edit operation only incurs four changes in sibling relationships: The one between b and b, and the one between b and e are destroyed. The sibling relationship between b and c, and the one between d and e are generated by the deletion operation.



B(T1)

B(T2)

Figure 3.3: Normalized Binary Tree Representation

As mentioned in 3.2.1, a binary tree corresponding to a forest retains all the structure information of the forest. Particularly, it gives a correspondence between trees and a special class of binary trees which have a root without right subtree. in the binary tree representation, the original parent-child relationships between nodes, except the ones between each inner nodes and its first child, are removed. The removed parent-child relationships are replaced by the link edges between the original siblings. This property makes the transformed binary tree representation appropriate for highlighting the effect of the edit-based operations on original trees. The novel algorithm proposed in this chapter are based on such observation and exploit the binary tree transformation properties, i.e. it store the structure information of trees by record the sibling relationship instead of all the parent-child relationship.

3.2.3 Vector Representation of Trees

To encode the structural information the transformed binary tree representation $B(T)$ of T is normalized as follows: In $B(T)$, for any node u, if u has no right (or left) child, a ϵ node (i.e., nodes with label ϵ do not exist in T) is appended as u's right (or left) child. This makes T a full binary tree in which all the original nodes have two children and all the leaves are with label ϵ (as shown in Figure 3.3). The normalized binary tree representation is defined as $B(T) = (N \bigcup \{\epsilon\},\ E_l,\ E_r,\ Root(B(T)),\ label)$, where ϵ denotes the appended nodes as well as their labels. To simplify the notation, in this chapter $u \in N$ represents the node as well as its label where no confusion arises. In order to quantify change detection in a binary tree, the concept *binary branch* on normalized binary trees is introduced:

Definition 3.2.2 (Binary Branch). *Binary branch* (or branch for short) is the branch structure of one level in the binary tree. For a tree T, $\forall u \in N$, there is a binary branch $BiB(u)$ in $B(T)$ such that $BiB(u) = (N_u,\ E_{u_l},\ E_{u_r},\ Root(T_u))$, where $N_u = \{u,\ u_1,\ u_2\}$ ($u \in N$; $u_i \in N \bigcup \{\epsilon\}$, $i = 1,\ 2$), $E_{u_l} = \{\langle u, u_1 \rangle_l\}$, $E_{u_r} = \{\langle u, u_2 \rangle_r\}$ and $Root(T_u) = u$ in the normalized $B(T)$.

According to the properties of normalized binary trees, we can have Lemma 3.2.3:

Lemma 3.2.3. *For each node $u \in N$ of a tree T, u may appear in at most two binary branches in the binary tree representation $B(T)$.*

PROOF:

1. *u can occur as root in at most one binary branch. This is obvious.*

2. *u can occur as the left (or right) child in at most one binary branch. u can not occur as the left child in one branch and as the right child in another branch at the same time; otherwise, u must have two parents in $B(T)$. That is contrary to the properties of trees.*

Assume that the universe of binary branches $BiB()$ of all trees in the dataset composes alphabet Γ and the symbols in the alphabet are sorted lexicographically on the string uu_1u_2. A representative vector of dimension $|\Gamma|$ can be built for each tree-structured data record, with each dimension recording the number of occurrences of a corresponding branch in the data. The formal definition of the binary branch vector is given in Definition 3.2.4.

Definition 3.2.4 (Binary Branch Vector). The *binary branch vector* $BRV(T)$ of a tree T is a vector $(b_1, b_2, \cdots b_{|\Gamma|})$, with each element b_i representing the number of occurrences of the ith binary branch in the tree. $|\Gamma|$ is the size of the binary branch space of the dataset.

To construct the binary branch vector of a tree, firstly an inverted file is built for all binary branches, as shown in Fig. 3.4(a). An inverted file has two main parts: a vocabulary which stores all distinct values being indexed, and an inverted list for each distinct value which stores the identifiers of the records containing the value. The vocabulary here consists of all existing binary branches in the datasets. The inverted list of each component records the number of occurrences of it in the corresponding trees. The resulting vectors of our transformation for the trees in Figure 3.1 and the normalized binary trees in Figure 3.3 are shown in Figure 3.4(b).

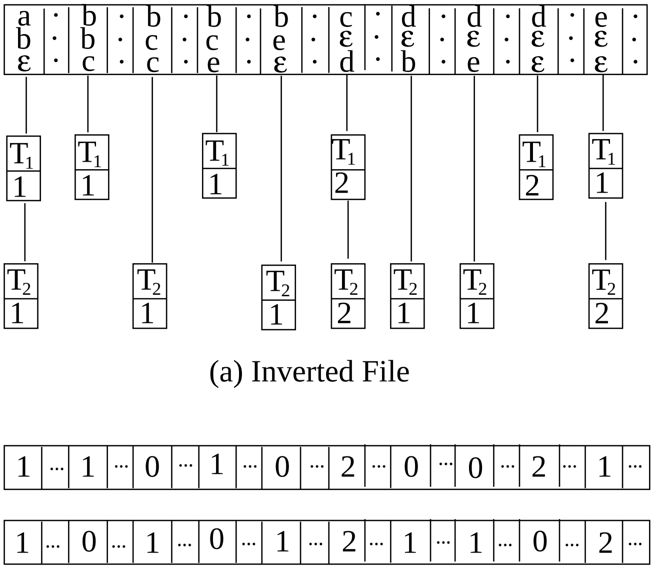
Based on the vector representation, a new distance of the tree structure can be defined as the L_1 distance between the vector images of two trees:

Definition 3.2.5 (Binary Branch Distance). Let $BRV(T_1) = (b_1,\ b_2,\ \cdots,\ b_{|\Gamma|})$, $BRV(T_2) = (b'_1, b'_2, \cdots b'_{|\Gamma|})$ be the binary branch vectors of trees T_1 and T_2 respectively. The binary branch distance of T_1 and T_2 is $BDist(T_1,\ T_2) = \Sigma_{i=1}^{|\Gamma|}|b_i - b'_i|$

The binary branch distance has the properties listed below: For all T_1, T_2 and T_3 in the dataset,

1. $BDist(T_1,\ T_2) \geq 0$, and $BDist(T_1,\ T_1) = 0$;

2. $BDist(T_1,\ T_2) = BDist(T_2,\ T_1)$;



(a) Inverted File

BRV(T_1)	1	···	1	···	0	···	1	···	0	···	2	···	0	···	0	···	2	···	1	···
BRV(T_2)	1	···	0	···	1	···	0	···	1	···	2	···	1	···	1	···	0	···	2	···

(b) Binary Branch Vectors

Figure 3.4: Binary Branch Vector Representation

3. $BDist(T_1,\ T_3) \leq BDist(T_1,\ T_2) + BDist(T_2,\ T_3)$.

Proof. The first two properties are obvious. For the third property, let $BRV(T_i) = (b_{i1},\ b_{i2},\ \cdots,\ b_{i|\Gamma|})$ for $i = 1,\ 2,\ 3$.

$$\begin{aligned} & BDist(T_1,\ T_2) + BDist(T_2,\ T_3) \\ & = \Sigma_{j=1}^{|\Gamma|}|b_{1j} - b_{2j}| + \Sigma_{j=1}^{|\Gamma|}|b_{2j} - b_{3j}| \\ & \geq \Sigma_{j=1}^{|\Gamma|}|b_{1j} - b_{3j}| \ = \ BDist(T_1,\ T_3) \end{aligned}$$

□

The third property means that the binary branch distance satisfies the triangular inequality. However, $BDist(T_1,\ T_2) = 0$ cannot imply that T_1 is identical to T_2. This is illustrated in Figure 3.5, where both trees have the same binary branch vector. So the binary branch distance is not a metric on tree-structured data.

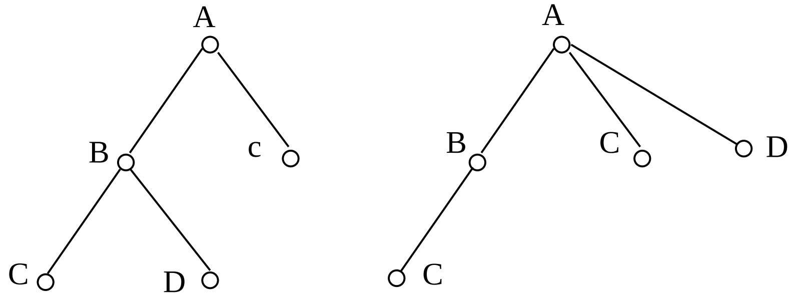


Figure 3.5: Trees with 0 Binary Branch Distance

3.2.4 Lower Bound of Edit Distance

In this section, the theoretical analysis of the new methods are given.

Theorem 3.2.6. *Let T and T' be two trees. If the tree-edit distance between T and T' is $EDist(T, T')$, then the binary branch distance between them satisfies the following: $BDist(T, T') \leq 5 \times EDist(T, T')$*

Proof. The theorem follows if it is proved that at most $5 \times k$ binary branch distance is incurred by k edit operations. Assume that edit operations $ed_1,\ ed_2,\ \cdots,\ ed_k$ transform T to T'. Accordingly, there is a sequence of trees $T = T_0 \rightarrow T_1 \rightarrow \cdots \rightarrow T_k = T'$, where $T_{i-1} \rightarrow T_i$ via ed_i for $1 \leq i \leq k$. Let there be k_1 relabeling operations, k_2 insertions and k_3 deletions. $k_1 + k_2 + k_3 = k$. It is sufficient to prove the theorem for one step of the transformation.

1. Assume that ed_i is a relabeling operation on some node v of the tree. According to Lemma 3.2.3, v occurs in at most two binary branches in $B(T_{i-1})$. Obviously, this operation retains the tree structure information of T_{i-1}. In these two branches, $label(v)$ is changed to the new one in the target tree $B(T_i)$. Assume that the count of the two binary branches in $BRV(T_{i-1})$ is in dimension l_1 and l_2, while the two new binary branches are in dimension l_3 and l_4. Then $BRV(T_{i-1})[l_m] - BRV(T_i)[l_m] = 1$, for $m = 1, 2$. $BRV(T_{i-1})[l_{m'}] - BRV(T_i)[l_{m'}] = -1$, for $m' = 3, 4$. So, $BDist(T_{i-1}, T_i) \leq 4$.

2. Assume that ed_j inserts a node v to transform T_{j-1} to T_j. Obviously, when v has a parent, a left sibling, a right sibling and child nodes, this operation leads to the maximum number of changes on the structure information. Figure 3.6 and Figure 3.7 demonstrate the insertion operation and the changes it causes on the binary tree representation. Let v be inserted under node v' and child nodes $w_{l+1}, \cdots w_{l+m}$ of v' in T_{j-1} become the child nodes of v in T_j.

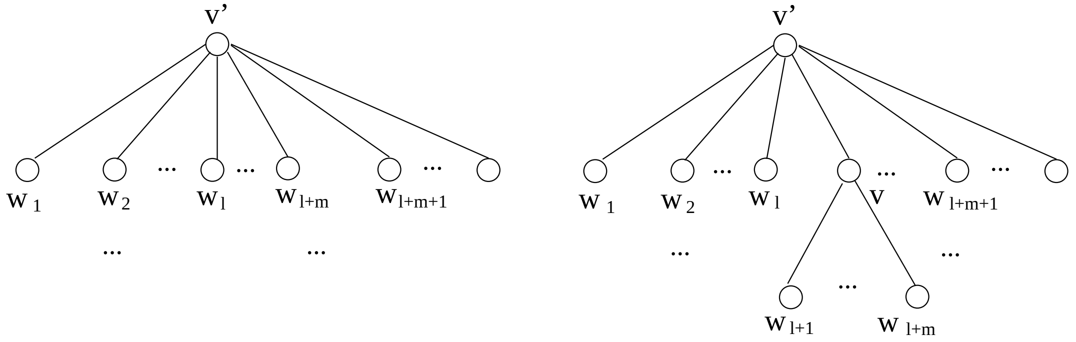


Figure 3.6: Insertion of Node v Under Node v'

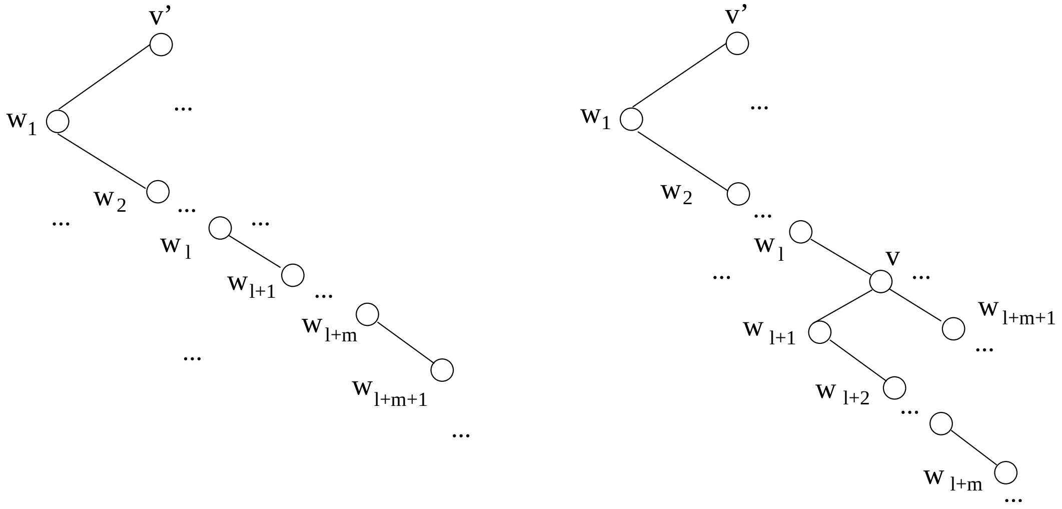


Figure 3.7: Changes of Binary Tree Incurred by Insertion

It is shown that at most five changes occur on the edges of $B(T_{j-1})$: Two edges $\langle v,\ w_{l+1}\rangle_l$ and $\langle v,\ w_{l+m+1}\rangle_r$ representing the structure information rooted on v are added into the binary tree. These edges comprise the binary branch $BiB(v)$.

So, assuming that it corresponds to dimension l in $BRV(T_j)$, then $BRV(T_j)[l] - BRV(T_{j-1})[l] = 1$. In addition, the sibling relationship between w_l and w_{l+1}, and between w_{l+m} and w_{l+m+1} in T_{j-1} (represented by $\langle w_l,\ w_{l+1}\rangle_r$ and $\langle w_{l+m},\ w_{l+m+1}\rangle_r$ respectively in $B(T_{j-1})$) are destroyed. This leads to the destruction of one of each binary branch $BiB_{T_{j-1}}(w_l)$ and $BiB_{T_{j-1}}(w_{l+m})$. [2] Thus, the values for the two corresponding dimensions in $BRV(T_j)$ are less than those in $BRV(T_{j-1})$ by 1. Finally, $\langle w_l,\ w_{l+1}\rangle_r$ is replaced by $\langle w_l,\ v\rangle_r$ in $B(T_j)$ for v is the right sibling of w_l after being inserted in T_j. $\langle w_{l+m},\ w_{l+m+1}\rangle_r$ is replaced by $\langle w_{l+m},\ \epsilon\rangle_r$ for w_{l+m} is the right most child of v in T_j after insertion. Then the values of the corresponding two dimensions, i.e., $BiB(w_l,\ *,\ v)$ and $BiB(w_{l+m},\ *,\ \epsilon)$, in $BRV(T_j)$ are larger than those in $BRV(T_{j-1})$ by 1 each. To sum up, $BDist(T_{j-1},\ T_j)$ is at most 5.

3. Deletion is complementary to insertion. Therefore the number of affected binary branches must be bounded by the same amount as for insertion.

According to the triangular inequality property of binary branch distance, we have

$$\begin{aligned} & BDist(T,\ T') \leq BDist(T_0,\ T_1) + BDist(T_1,\ T_2) + \cdots + BDist(T_{k-1},\ T_k) \\ & \leq 4 \times k_1 + 5 \times k_2 + 5 \times k_3 \leq 5 \times k \\ & \leq 5 \times EDist(T,\ T'). \end{aligned}$$

□

3.2.5 Extended Study

As shown above, the generalized analysis is similar to that of the q-gram method [110] for solving the k-difference problem of strings. The number of occurrences of each q-gram (i.e., all strings of length q over the alphabet) in any two strings are counted. If two

[2] The $*$ can be any label in Σ or the label ϵ.

strings are similar, they have many q-grams in common. Formally, if the edit-distance of strings S_1 and S_2 is k, then they have at least $\max(|S_1|, |S_2|) - (k-1)q - 1$ q-grams in common. When applied to similarity search problems in which the full strings are involved, the q-gram method usually trades off the false positive for the false negative rate by adjusting the length of the q-gram searched [57]. Binary branches can be viewed as playing the role of q-gram structures for tree data. The vector images of trees can be extended to record multiple level binary branch profiles. Firstly, the formal definition of the q-level binary branch is given below:

Definition 3.2.7 (q-level Binary Branch). The q-level binary branch $BiB\_q(n_0,\ n_1,\ \cdots,\ n_{2^q-2})$ is the perfect binary tree of height $q-1$, where $n_0,\ n_1,\ \cdots,\ n_{2^q-2}$ is the sequence obtained by preorder traversing the perfect binary tree (with all leaf nodes at the same depth and all internal nodes having degree 2).

The binary branch defined in the previous section is indeed the two-level binary branch. Similar to the computation of q-grams for strings, our sliding window is a perfect binary tree with height $q-1$ (i.e., all leaves are of the same depth $q-1$). The sliding window shifts one level each time along the path from the root to the leaves. For each node u in the tree, there is a q-level binary branch rooted at u in the binary tree representation consisting of the perfect binary subtree rooted at n. If the subtree of height $q-1$ rooted at u is not a perfect binary tree in the transformed representation, ϵ-nodes can be appended to complete it.

The multiple level binary branch is used to maintain structures of fixed size and fixed shape in the original data. Obviously, it encodes more information than the two-level binary branch. We can extend the binary branch vector to the *characteristic vector* $BRV\_q(T)$, which includes all the elements in the q-level binary branch space. The q-level binary branch distance $BDist\_q(T,\ T')$ is defined as the L_1 vector distance between the images of the trees T and T' under the q-level mapping. Figure 3.8 shows the

3-level binary branch of the T_1 tree in Fig 3.1.

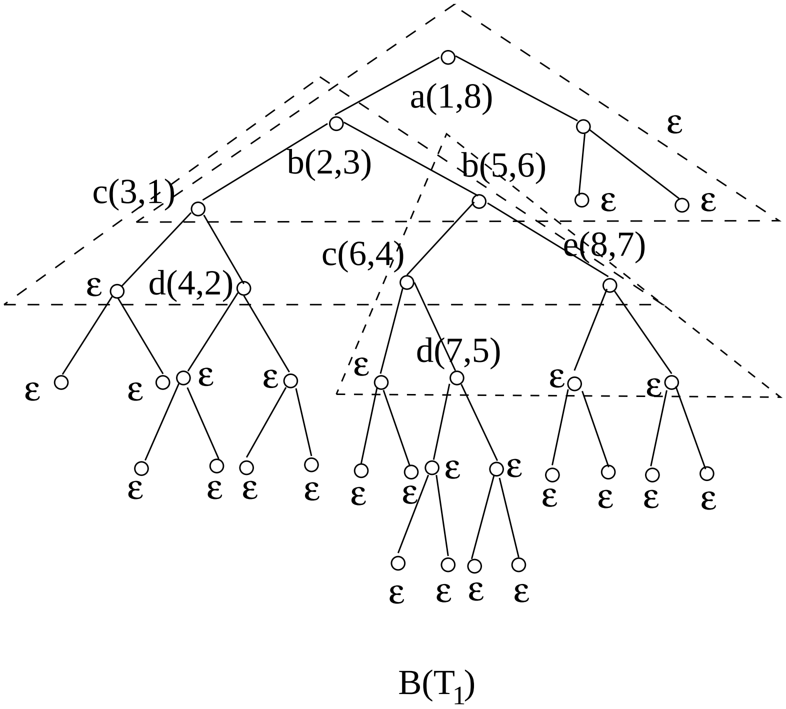


Figure 3.8: 3-level Binary Branch Vector Examples

Theorem 3.2.8. *Let T and T' be two trees. If the tree-edit distance between T and T' is $EDist(T,T') = k$, and the corresponding edit operation sequence consists of k_1 relabeling operations, k_2 insertions and k_3 deletions, then the q-level binary branch distance between them $BDist\_q(T,T') \leq [4 \times (q-1)+1] \times k$*

Proof. The proof methods here are similar to those of Theorem 3.2.6.

It is sufficient to consider the case when the tree edit-distance between T and T' is 1. Let $Anc(n,i)$ denote the lowest ith ancestors of node n. Let $Path(n_1,n_2)$ be the path from node n_1 to node n_2 in the tree, while $PathLen(n_1,\ n_2)$ be the length of $Path(n_1,\ n_2)$, i.e., the number of parent-child edges between node n_1 and n_2.

1. Assume that T' is obtained from T by relabeling a node v. It is obvious that each node in the tree T appears in at most q q-level binary branches. These

are the ones rooted at the nodes $Anc(n, q-1)$ and the one rooted at v itself if $PathLen(Root(B(T))) \geq (q-1)$. For example, the triangles of the dashed line in Figure 3.8 show the 3 3-level binary branches $b(5,6)$ appears. Then the relabeling of node v destroys at most q q-level binary branches. At the same time, it generate the same number of q q-level binary branches in $B(T')$, one for each of the destroyed ones. This leads to at most $2 \times q$ q-level binary branch distance between T and T'.

2. Assume that T' is obtained from T by inserting a node v under node v' as shown in Figure 3.7. Just as analyzed in Theorem 3.2.6, insertion of a node in the tree leads to the destroy of at most two edges between parents and their right child nodes in the transformed binary tree ($< w_l,\ w_{l+1} >_r$ and $< w_{l+m},\ w_{l+m+1} >_r$) and generate two new ones ($< w_l,\ v >_r$ and $< w_{l+m},\ \epsilon >_r$) for replacement. One edge in the binary tree exits in $(q-1)$ q-level binary branches. So these changes leads to 4 difference between the q-level binary branch vectors of T and T'. In addition, the relationship between the inserted node v and it's first child and it's next sibling are added into the transformed binary tree of T': $< v,\ w_{l+1} >_l$ and $< v,\ w_{l+m+1} >$. These two edges consists of a binary branch $BiBranch(v,\ w_{l+1},\ w_{l+m+1})$. This binary branch occurs in $(q-1)$ q-level binary branches. However, except the one rooted at v, all these q-level binary branch also contains $< w_l,\ v >_r$. So the difference of the value on the dimensions of these $(q-2)$ q-level binary branches are already counted. Only the the value of dimension of the q-level binary branch which is rooted at v is increased by 1. So, one insertion operation on any node in the tree T to generate T' cause at most $[4 \times (q-1) + 1]$ q-level binary branch distance.

3. The deletion is complementary to insertion. So each deletion operation cause at most $[4 \times (q-1) + 1]$ q-level binary branch distance too.

From the above analysis, we obtain:

$$BranchDist\_q(T,\ T') \leq 2k_1 q + k_2[4(q-1)+1] + k_3[4(q-1)+1] \tag{3.1}$$

Since $q \geq 2$, $BranchDist\_q(T,\ T') \leq [4(q-1)+1]k$ □

The binary branch distance $BDist\_q(T_1, T_2)$ increases as the level of the binary branch q increases. This is due to the fact that the higher the level is, the more information of the tree structure is encoded in the binary branches. At one extreme, q is equal to the height of the normalized transformed binary tree; then all the structural information of the original tree is encoded. However, in such a situation, the filter algorithm is of no use. At the other extreme q is equal to 1; in this case, the filter efficiency is too low. We do not discuss this option as it records no structure information of the original tree at all. According to Theorem 3.2.8, $BDist\_q(T,\ T')/[4(q-1)+1]$ can be used as a series of approximations for the tree-edit distance with different resolutions. So the level q of the binary branch can be adjusted to improve filter efficiency when solving the similarity search problem.

3.3 Enhancement of Similarity Search on Tree-structured Data

In the previous section, the tree structures and the tree edit distance metric are mapped to a numeric vector space and the L_1 norm distance. Although, according to its properties, the binary branch distance is not a metric, it approximates and lower bounds the tree-edit distance metric. Just as q-gram methods can be used to speed up similarity search for strings, the distance-embedded lower bounds can be integrated into the filter-and-refine framework to speed up similarity search by reducing the number of expensive similarity distance computations. This section presents the new filter-and-refine algorithm for processing similarity search on the tree-structured data by exploiting the lower bounds.

3.3.1 Basic Algorithm

Similarity search on various data usually refers to range queries and k nearest neighbor queries. Range queries find all objects in the database which are within a given distance τ from a given object; k nearest neighbor (k-NN) queries find the k most similar objects in the database which are closest in distance to a given object. Other types of search can be composed by these two similarity queries. When searching tree-structured data, similarity is measured by tree-edit distance.

As mentioned in Chapter 1, the similarity evaluation of large trees in massive datasets based on tree-edit distance is a computationally expensive operation. Traditionally, the filter-and-refine architecture is utilized to reduce real distance computation by employing the lower bounds of the real distance [95]: In the first step (i.e., filtration), objects that cannot qualify are filtered out. In the second step (i.e., refinement), verification of the original complex similarity measure is necessary only for the candidates filtered through. The objects satisfying the query predicate are reported as results. The completeness of the results is guaranteed by the lower bound property: If the lower bound distance is greater than the query range, it is safe to filter out the data since its real edit distance cannot be less than that range.

The new method proposed in this chapter is to embed an easy-to-compute distance function that is the lower bound of the actual tree edit distance into the filter-and-refine framework. The optimistic bound used by the similarity search is based on the binary branch vector distance of the trees. In addition to the number of occurrences of individual binary branch, the positional information of the binary branch is also important in exploring the structure information of the trees. In the description of this chapter, the two-level binary branch is used. However, the approach can be easily generalized to q-level binary branches.

3.3.2 Optimistic Distance for Similarity Queries

The efficiency of the filter-and-refine architecture is based on the hypothesis that the lower bound function is much quicker to evaluate than real distance. As shown in Section 3.2.4, binary branch distance lower bounds edit distance effectively: The lower bound function can be computed in $O(|T| + |T'|)$ time, which is much more succinct than edit distance computation. So, using binary branch distance as optimistic bound can reduce the overall processing time. Like using the q-gram methods to solve the approximate string matching problem, not only the occurrences of the q-grams, but also their positions can be exploited to measure the similarity of the pattern and certain subsequence of the strings [102, 47]. The idea is that: given two strings with distance less than l, two identical q-grams in the two strings respectively cannot be matched if their positions differ by more than l. Otherwise, more than l symbols have to be inserted or deleted. The size of the corresponding series edit operations must be larger than l.

Binary branch filtration also exhibits this property. First, a proposition is given as follows:

Proposition 3.3.1. Let the edit distance of T_1 and T_2 be less than l. Each node u in the trees is numbered by its preorder traverse position (or postorder traverse position). In the mapping corresponding to the edit distance, the node $u \in T_1$ cannot be mapped to $v \in T_2$ if the difference of the numbers of u and v is larger than l.

Proof. In the preorder traversal numbering, the numbers which are smaller than that of u are assigned to ancestors of u or the nodes that are to the left of u, while the ones that are larger than that of u are assigned to descendants or the nodes to the right of u. Since the edit operation mapping preserves sibling order and ancestor order, if the two nodes are matched, and their number difference is larger than l, then there must be more than l deletions or insertions. This is contrary to the premise that the edit distance is less than l.

For the postorder traverse position, the numbers which are smaller than that of u are assigned to descendants of u or the nodes that are to the left of u, while the ones that are larger than that of u are assigned to ancestors or the nodes to the right of u. If the number of u is l more (or less) than that of v, than there are more than l nodes under (or above) u or to the left (right) of u. Similarly, there must be more than l deletions or insertions. This is contrary to the premise that the edit distance is less than l. □

For each binary branch $BiB(u, u_1, u_2)$, the positional structure is defined, denoted as $(BiB(u, u_1, u_2),\ pre(u),\ post(u))$, where $pre(u)$ and $post(u)$ are the preorder and the postorder traversal positions of u in T respectively (resp. the preorder traverse and inorder traverse of $B(T)$). Based on the positional binary branch, the mapping $M(T_1, T_2, pr)$ between the positional binary branches of T_1 and T_2 is defined with positional range pr, which is any set of pairs of positional binary branches (($BiB(u, u_1, u_2)$, $pre(u)$, $post(u)$), ($BiB(v, v_1, v_2)$, $pre(v)$, $post(v)$)) satisfying:

1. the mapping is one-to-one;

2. $BiB(u, u_1, u_2) = BiB(v, v_1, v_2)$;

3. $|pre(u) - pre(v)| \leq pr$ and $|post(u) - post(v)| \leq pr$.

Given two trees T_1 and T_2 with $EDist(T_1,\ T_2) \leq l$. For two positional binary branch $(BiB(u, u_1, u_2),\ i_1,\ i_2)$ and $(BiB(u, u_1, u_2), i'_1, i'_2)$ in T_1 and T_2 respectively, if the maximum positional differences $\max(|i_1 - i'_1|,\ |i_2 - i'_2|) > l$, then the two binary branches $BiB(u, u_1, u_2)$ in the two trees cannot be mapped to each other in the mapping leads to the minimum number of edit operation.

For the example in Fig. 3.3, the numbering beside each node is the position specification of the corresponding binary branch. Then, the positional binary branches of T_1 in Fig. 3.3 is: ($(BiB(a, b, \epsilon), 1, 8)$, $(BiB(b, c, b), 2, 3)$, $(BiB(c, \epsilon, d), 3, 1)$, $(BiB(d, \epsilon, \epsilon), 4, 2)$, $(BiB(b, c, e), 5, 6)$, $(BiB(c, \epsilon, d), 6, 4)$, $(BiB(d, \epsilon, \epsilon), 7, 5)$, $(BiB(e, \epsilon, \epsilon), 8, 7)$). And that of T_2 are ($(BiB(a, b, \epsilon), 1, 9)$, $(Bib(b, c, c), 2, 5)$, $(BiB(c, \epsilon, d), 3, 1)$, $(BiB(d, \epsilon, b), 4, 2)$, $(BiB(b, e, \epsilon), 5, 4)$, $(BiB(e, \epsilon, \epsilon), 6, 3)$, $(BiB(c, \epsilon, d), 7, 6)$, $(B\,iB(d, \epsilon, e), 8, 7)$, $(BiB(e, \epsilon, \epsilon), 9, 8)$); Assume the positional range $pr = 1$. It is obvious that $(BiB(c, \epsilon, d), 3, 1)$ in T_1 can only be mapped to $(BiB(c, \epsilon, d),\ 3, 1)$ in T_2; While $(BiB(c, \epsilon, d), 6, 4)$ and $(BiB(c, \epsilon, d), 7, 6)$ cannot be mapped to each other. $(BiB(e, \epsilon, \epsilon), 8, 7)$ in T_1 can be mapped to $(BiB(e, \epsilon, \epsilon), 9, 8)$ in T_2, but cannot be mapped to $(BiB(e, \epsilon, \epsilon), 6, 3)$.

For two trees T_1 and T_2, we denote the maximum-sized mapping as $M_{max}(T_1, T_2, pr)$. The subset of it which is related to a given binary branch $BiB \in \Gamma$ is denoted as $M'_{max}(T_1, T_2, BiB, pr)$. Obviously, $M'_{max}(T_1, T_2, BiB, pr)$ is the maximum-sized mapping on the binary branch BiB. Given the preorder and postorder position sequences of BiB in T_1 and T_2 in ascending order, $|M'_{max}(T_1, T_2, BiB, pr)|$ (size of $M'_{max}(T_1, T_2, BiB, pr)$) can be computed in linear time. A new distance between two trees can be defined based on $|M'_{max}(T_1, T_2, BiB, pr)|$:

Definition 3.3.2 (Positional Binary Branch Distance). Given two trees T_1 and T_2, their binary branch vectors $BRV(T_i) = (b_{i1}, b_{i2}, \cdots, b_{i|\Gamma|})$ (, where $i = 1, 2$) and the positional range specification pr, the positional binary branch distance with range pr is

$$PosBDist(T_1, T_2, pr) = \sum_{j=1}^{|\Gamma|} (b_{1j} + b_{2j} - 2|M'_{max}(T_1, T_2, j, pr)|)$$

Proposition 3.3.3. If $PosBDist(T_1, T_2, l) > 5 \times l$, then $EDist(T_1, T_2) > l$.

Proof. We prove the contrapositive proposition: If the edit distance is less than l, then

$PosBDist(T_1, T_2, l) \leq 5 \times l$. According to the definition of positional binary branch, $PosBDist(T_1, T_2, l)$ differs from $BDist$ in that, in T_1 and T_2, it does not match the same binary branches whose position differences are larger than l. For any positional binary branch $(BiB(u, u_1, u_2), pre(u), post(u))$, if there is no element in $M_{max}(T_1, T_2, l)$ that corresponds to it, the node u should be changed by some edit operation. According to Definition 3.3.2, the positional binary branch distance is the sum of the differences on the binary branches incurred by the edit operations to change T_1 to T_2. And according to Theorem 3.2.6, one edit operation changes at most 5 binary branches. Thus $PosBDist(T_1, T_2, l) \leq 5 \times l$. □

Obviously the positional binary branch distance is related to the positional range specification. Theoretically, the positional range for two trees T_1 and T_2 can increase from $pr_{min} = 0$ to $pr_{max} = |T_1| + |T_2|$ and the positional binary branch distance decrease correspondingly. Given $pr = pr_{min}$, the corresponding positional binary branch distance computed has the maximum possible value:

$$PosBDist(T_1, T_2, pr_{min}) = PosBDist_{max}$$

, computed by matching only the identical binary branches which have the same positions. Apparently,

$$PosBDist_{max}/5 > pr_{min}$$

Given $pr = pr_{max}$, the corresponding positional binary branch computed has the minimum possible value

$$PosBDist(T_1, T_2, pr_{max}) = PosBDist_{min} = BDist(T_1,\ T_2)$$

It is obvious that

$$PosBDist_{min}/5 \leq EDist(T_1,\ T_2) \leq pr_{max}$$

Then, there must be a given positional range pr_i s.t. $pr_{min} \leq pr_i \leq pr_{max}$ which is the maximum positional range that satisfies $PosBDist(T_1, T_2, pr_i)/5 > pr_i$. According to the analysis of Proposition 3.3.3, $EDist(T_1,\ T_2) \geq (pr{+}1)$, where $pr = pr_{min}, \cdots, pr_i$. Thus, $(pr_i + 1)$ is a lower bound of edit distance. Note that $BDist(,)$ is the minimum value for $PosBDist$ and that for $pr_i + 1$, $PosBDist(T_1, T_2, pr_i + 1)/5 \leq (pr_i + 1)$, so we have:

$$BDist(T_1, T_2)/5 \leq PosBDist(T_1, T_2, pr_i + 1)/5 \leq (pr_i + 1)$$

Thus, $pr_i{+}1$ is a closer lower bound of edit distance between T_1 and T_2 than $BDist(T_1,\ T_2)/5$. A better optimistic bound, pr_{opt}, of the edit distance can be obtained by searching the minimum value of the positional range pr_i ($pr_{min} \leq pr_i \leq pr_{max}$) satisfying

$$PostBDist(T_1, T_2, pr_i)/5 \leq pr_i$$

In practice, we can reduce the search range further. Since $EDist(T_1, T_2) \geq ||T_1| - |T_2||$, $pr_{min} = ||T_1| - |T_2||$. At the same time, it is meaningless to set the pr_{max} to be larger than $max(|T_1|, |T_2|)$;

3.3.3 Similarity Search Algorithm

This section gives the algorithm for constructing vectors and a novel filter-and-refine algorithm for similarity search utilizing the positional binary branch distance. The steps of vector construction is shown in Algorithm 1.

Algorithm 1 *vector construction*

```
Input:
The data set $D$
Output:
The vector representations of the data $BRV$,
The preorder positions $preOrderPos$,
The postorder positions $postOrderPos$,
initialize the inverted file index $IFI$ to be empty;
for each record $T \in D$ do
    $PrePosition = 0$;
    $PostPosition = 0$ ;
    $Traverse(Root(T), PrePosition, PostPosition, IFI)$;
$l = 0$;
for each entry $i$ in $IFI$ do
    for each entry $j$ in the inverted list of $i$ do
        $k = IFI[i][j].Tid$;
        $BRV[k][l].Bib \leftarrow i$;
        $BRV[k][l++].Count \leftarrow IFI[i][j].occurrence$;
        Build positional sequence $preOrderPos[k]$;
        Build positional sequence $postOrderPos[k]$;
```

Function: $Traverse(R, \& Preorder, \& Postorder, IFI)$

```
construct binary branch $BiB_R$ of $R$ by calling
$getFirstChild(R)$ and $getNextSibling(R)$;
{two level binary branch}
$Preorder$++;
$insertPreOrder(Tid, BiB_R, IFI, Preorder)$;
for each child node $r_i$ of R do
    $Traverse(r_i, Preorder, Postorder, IFI)$;
$Postorder$++;
$insertPostOrder(Tid, BiB_R, IFI, Postorder)$;
```

In the vector construction algorithm, an extended inverted file IFI is utilized to build the vector representation. The inverted list of each binary branch records the data record Tid, the number of occurrences of this branch and the respective positions at which it appears in the corresponding data. Firstly, each tree-structured data is recursively traversed and the IFI is constructed by calling the function $Traverse(\ )$ to obtain the binary branch information in Figure 3.3. In the function, the binary branch BiB_R of the current node is built by calling the $getFirstChild()$ and $getNextSibling()$ functions of the parser. Then, in function $insertPreOrder(\ )$, the corresponding entry of BiB_R in IFI is found by some hashing function. Then the component for data T (identified by Tid) at the end of inverted list is updated: The number of occurrences is increased by 1. The preorder position of the branch is recorded. In function $insertPostOrder(\ )$, the postorder position is recorded.

After the construction of IFI in $Traverse(\ )$, the sparse vector representation of each data are built by scanning IFI (in Line 7-13 of Algorithm 1): For each branch that occurs in the data, the id of the branch and the number of its occurrences is recorded in the vector. In addition, two arrays recording the branch positions (for preorder and postorder respectively) are constructed from IFI. Both are sorted according to the branches and in ascending order. The positions are stored according to the binary branch id. And for each binary branch, the positions are stored in ascending order in the two sequence.

The procedure for k-NN search is shown in Algorithm 2. First, the query T_q is preprocessed to construct the vector representation and position sequences. The process is similar to $Traverse(\ )$ except that the inverted file need not to be built. In line 3 of Algorithm 2, the optimistic bound of the distance between the query and each data object is computed by calling function $SearchLBound(\ )$; The steps of function $SearchLBound(\ )$ is shown in Algorithm 3.3.3. In the function, the optimistic bound is searched for in the range

$$[diff(size(vec_{T_q}), size(BRV[i])),\ max(size(vec_{T_q}),\ size(BRV[i]))]$$

. Since the search range is ordered, we use the binary search algorithm. In line 3 and line 8 of Function $SearchLBound(\ )$, the distance $PosBDist()$ are computed based on $|M'_{max}()|$.

After the optimistic bounds of all the vectors are obtained, at line 4 of the Algo-

Algorithm 2 *k-NN on tree-Structured data*

```
Input: The data set $D$,
The vector representation of data $BRV$,
The preorder positions $preOrderPos$,
The postorder positions $postOrderPos$,
The query $T_q$;
Output:
The result set of $k$ nearest neighbors of $T_q$
1: construct vector and position arrays $vec_{T_q}$,
   $preOrderPos_{T_q}$ and $postOrderPos_{T_q}$ for $T_q$;
2: for each vector $i$ in $BRV$ do
3:     $LowerBound[i] = SearchLBound(vec_{T_q}, BRV[i],$
       $preOrderPos[i], postOrderPos[i], preOrderPos_{T_q}, postOrderPos_{T_q})$;
4: sort the $LowerBound$ and $D$ into $LowerBound'$ and $D'$ in ascending order of the
   lower bound distances;
5: initialize the max heap $KNN$, s.t. capacity(KNN)=k;
6: for i From 0 To $|D|$ do
7:     if ($KNN.size = k$)AND($LowerBound'[i] > KNN[0].key$) then
8:         BREAK;
9:     Retrieve the corresponding data $T_i$;
10:    $editDist = EDIST(T_i, T_q)$;
11:    if $KNN.size$ is less than $k$ then
12:        insert $T_i$ with the key $editDist$ in $KNN$ ;
13:    else
14:        pop up $KNN[0]$;
15:        insert and push down $T_i$ with the key $editDist$ in $KNN$;
16: return $KNN$;
```

rithm 2, the $LowerBound$ array and the data tree id are sorted in ascending order of the optimistic bounds to ensure that vectors of high possibility in being the results are processed before others. Second, the pruning procedure of traditional filter-and-refine similarity search steps are adopted [95, 8, 77, 83] to reduce real distance computation. A max heap KNN of capacity k is used to facilitate query processing. $KNN[].key$s are the real edit distance of the current results. $KNN[0].key$ has the maximum value and it is the pessimistic bound. If the optimistic bound of the next vector is smaller than the pessimistic bound, the data T_i associated with this vector need to be retrieved and the real edit distance is evaluated (line 10). The real distance is used to update the

```
**Function:**$SearchLBound(vec_{T_q}, BRV_i, preOrderPos_i, postOrderPos_i, preOrderPos_{T_q}, postOrderPos_{T_q})$
$pr_{min} = diff(size(vec_{T_q}), size(BRV_i))$;
$pr_{max} = max(size(vec_{T_q}), size(BRV_i))$;
$PosBDist_{max} = PosDiff(vec_{T_q}, BRV_i, preOrderPos_i, postOrderPos_i, preOrderPos_{T_q}, postOrderPos_{T_q}, pr_{min})$;
**if** $PosBDist_{max}/5 \leq pr_{min}$ **then**
    Return $pr_{min}$;
**while** $pr_{min} \leq pr_{max}$ **do**
    $pr_{half} = (pr_{min} + pr_{max})/2$;
    $PosBDist = PosDiff(vec_{T_q}, BRV_i, preOrderPos_i, postOrderPos_i, preOrderPos_{T_q}, postOrderPos_{T_q}, pr_{half})$;
    **if** $PosBDist/5 \leq pr_{half}$ **then**
        $pr_{max} = pr_{half} - 1$;
    **else**
        $pr_{min} = pr_{half} + 1$;
Return $pr_{half} + 1$;
```

current result as well as the pessimistic bound. This process continues till the optimistic bound of the next vector is larger than the pessimistic bound. Then the query processing ends. It is impossible for the remaining data to be closer to the query than the results for their lower bounds are already larger than the maximum distance between the query and current results.

Range query processing is similar to k-NN query processing; The difference is that there is a specified range τ for the query. According to Proposition 3.3.3,

$$\max(PosBDist(T, T_q, \tau)/5,\ pr_{opt})$$

should be considered as the optimistic bound in the filtering step. If it is larger than τ, the corresponding data cannot be the result and should be pruned accordingly.

3.3.4 Complexity Analysis

In this section, the time and space complexities analysis of the vector construction method and optimistic bound computation method is given. In order to calculate running time complexity, each step of the algorithm is considered. Assume that the size of the dataset, i.e., the total number of tree data objects, is $|D|$. For record T_i, there are $|T_i|$ nodes in it. The vocabulary of inverted file IFI is implemented by one hashing function. According to Algorithm 1, function $Traverse()$ is called recursively to traverse each node and insert the binary branch information of the current node into $IFI.$ Each time the new entries are appended at the end of the inverted list. So each update of IFI is of constant time complexity. Thus, the IFI construction is of linear complexity. As we store in IFI only the existing vocabulary of the dataset, the worst case is that all the nodes in the datasets have got different binary branches. Thus, the size of the vocabulary is at most $\sum_{i=1}^{|D|}|T_i|$. In addition, each node in each tree has one corresponding entry in the inverted list. In total, the space complexity of IFI is also $O(\sum_{i=1}^{|D|}|T_i|)$. To build the vector representation, the whole IFI has to be scanned once. So the time and space complexities of the whole vector construction algorithm are both $O(\sum_{i=1}^{|D|}|T_i|)$.

Next, we analyze the optimistic bound computation complexity in our query processing method. Given one query T_q, we need to compare its vector and its positional sequence with those of each data T_i. As mentioned in section 3.3.3, we use the binary search algorithm to obtain the optimistic bound between $||T_i|-|T_q||$ and $max(|T_i|,|T_q|)$. Each search process is of linear complexity $O(|T_i| + |T_q|)$. Then the time complexity for this step is $O(\sum_{i=1}^{|D|}(|T_i| + |T_q|) \times log(min(|T_i|,|T_q|)))$, and the space complexity is $O(\sum_{i=1}^{|D|}|T_i| + |T_q|)$.

3.4 Experimental Results

In this section, the performance comparison of the new filter-and-refine similarity search algorithm which integrates binary branch distance and the lower bound of edit distance (denoted as $BiBranch$ in Figure 3.9 through 3.17) against the histogram filtration methods proposed in [56] (denoted as $Histo$ in those figures). The set of experiments were done on synthetic datasets to show the algorithms' sensitivity to different features of the data. The experiments on real dataset show the algorithms' performance on different query characteristics. Finally, the effect of level q on the algorithm is discussed. All the experiments are conducted on a workstation with Intel Pentium IV 2.4GHz CPU and 1GB of RAM. And the the novel algorithm and the algorithms proposed in [56] are implemented in C++.

The synthetic data generator is similar to that of [122], except that the simulation of the website browsing behavior is not necessary, but instead the data distance need to be controlled. The program constructs a set of trees based on specific parameters. Four groups of parameters, the fanout of tree nodes, the size of trees, the number of labels and the edit operations are all random variables conforming to some distributions. The fanout and the size of the trees are sampled from normally distributed values, denoted by $N\{x_1, x_2\}$, where x_1 and x_2 are the mean and standard deviation of the normal distribution. The number of labels in the dataset is denoted by Ly, where y is its value. Multiple nodes in each tree can share the same label. For example, the specification $N\{4, 0.5\}N\{50, 2\}L8$ means that in the generated trees, the fanout of nodes conforms to normal distribution with mean 4 and variance 0.5. The total number of nodes in each tree conforms to normal distribution with mean 50 and standard deviation 2. And there are eight labels in the whole dataset. We also use another parameter Dz, the decay factor, to explicitly specify the distribution of the edit operations. The generator consists of the following steps: Firstly, a given number of seeds of the dataset are generated according to the first three groups of parameters. At the beginning of each seed generation, the maximum size is randomly sampled from $N\{50, 2\}$. Then, the tree grows by breadth first processing. The label of current node is sampled uniformly from the eight labels. Next, we check whether the current size of the tree exceeds the maximum size. If so, the process terminates. Otherwise, the number of children of current node is sampled from $N\{4, 0.5\}$. Secondly, new tree is generated from one of the seeds by changing each node of it with the probability specified by Dz. The changes are equiprobably insertion, deletion, and relabeling. The data generated from the seeds is used as the seed for the next data generation. In our experiments, we adopted 0.05 as the decay factor. Experiments with other settings had similar results.

For the real datasets, we used $DBLP$, which consists of bibliographic information on major computer science journals and proceedings. It is of XML document format and includes very bushy and shallow trees in the repository. The average depth is 2.902, and there are 10.15 nodes on average in each tree.

In each experiment, 100 queries were randomly selected from the dataset. The results shown in this chapter were all averaged on the queries. CPU time consumption is one performance measure. As real edit distance computation is the most costly part of similarity search on tree-structured data, the percentage of data which are not filtered out and for which the real distances have to be evaluated is an important measure of the algorithm efficiency. It is defined as:

$$\left(\frac{|True\ Positive| + |False\ Positive|}{|Dataset|}\right) \times 100\%$$

Timings were based on processor time. As the source code of histogram filtration was not available, for time consumption, we compared our filter-and-refine algorithm with the sequential search algorithm.

For the histogram filtration algorithm, three types of histogram vectors are used: One histogram records the distribution of heights of every node in the tree, a second records the fanouts for each of the nodes, and a third records the distribution of labels used. As mentioned in section 3.3.3, in the binary branch vector, only the non-zero dimension is stored. Also, the positional information for binary branches is stored for each node which equals to the size of the trees. To use equal amount of space, we set the sum of dimension of the three type histogram vectors for one tree to be the averaged vector size plus two averaged tree size in a given dataset.

3.4.1 Sensitivity Test

In the first set of experiments, a series of sensitivity analysis to the parameters of the dataset is carried out. The first three arguments of the data generator were set with different distributions. All the datasets generated included 2000 trees. Figure 3.9 to Figure 3.13 show the relative performance of the methods for various parameter settings. They compare the percentage of accessed data for the binary branch filtration and the histogram filtration (shown as the bars in the figures) and the CPU time consumption of the binary branch filtration and the sequential search (shown as the lines). The results shown are for range queries as well as k-NN queries. Each range was set to be the 1/5 of the average distance among the whole datasets. For k-NN queries, we retrieved 0.25$\%$ of the trees of the dataset.

Figure 3.9 and Figure 3.10 illustrate the performance of the two algorithms when the fanout varied. The mean values of it in the four datasets increased from 2 to 8 with the variance fixed to be 0.5. In order to analyze the effect of fanout, we diminished the effect of tree size and label number. The mean values of the tree size in the four datasets were all 50, and the standard deviation was limited to 2. Thus most trees in the datasets should have a size range from 46 to 54. The label number for each dataset was fixed at 8. It is shown that the binary branch filtration accessed at most 3.35% of the number of data objects accessed by the histogram filtration for the range queries and at most

23.08% for the k-NN queries. When fanout was 2, both filtration methods accessed the most data. The reason is that the probability that the fanout of nodes is 0 is much higher when the mean is set to be 2. Then the structure distance in this dataset is larger since the variation of height is larger than other sets. When the fanout is increased to 4, the height difference becomes much less. We also see that with increasing fanout, the histogram filtration accessed less data for range queries. This is because degree histogram yields better filtration power for larger fanout. However, for the k-NN queries, similar trends did not appear since the mean of the real distance increased as the fanout increased, and the search radius had to grow to retrieve the k most similar data [24]. In Figure 3.10, for the binary branch filtration, when the access rate is only 1.4%, the time consumption of binary branch distance evaluation is only 1.92% of the CPU cost of sequential query processing. This is consistent with the theoretical analysis that real distance consumption is overwhelming. So the extra costs incurred by the filtering can be ignored.

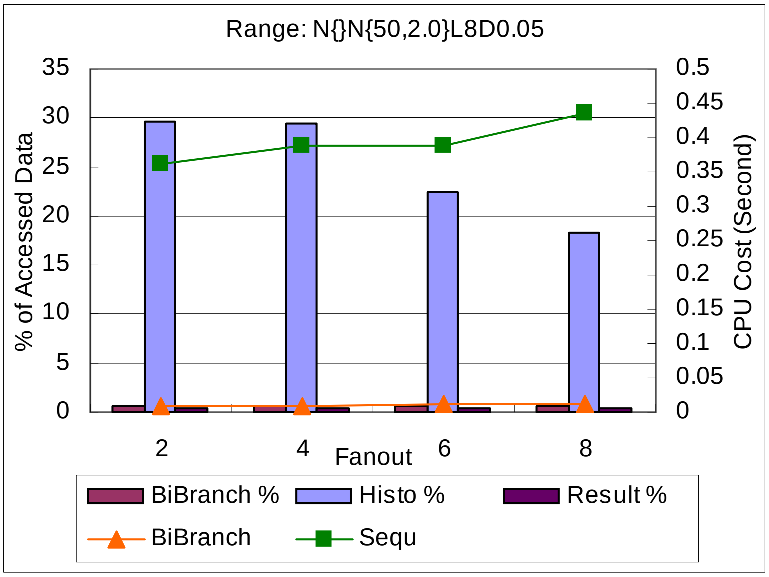


Figure 3.9: Sensitivity to Fanout Variation for Range Queries

Figure 3.11 and Figure 3.12 show the percentage of accessed data and CPU cost when the mean size of trees varied. The results of the k-NN queries are similar to that of the

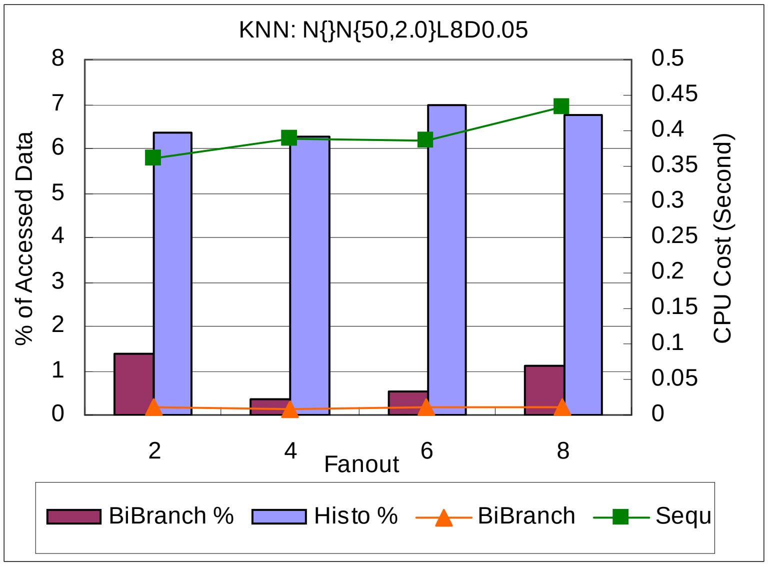


Figure 3.10: Sensitivity to Fanout Variation for k-NN Queries

range queries. In these experiments, the fanout of the datasets conformed to $N\{4, 0.5\}$. The label size is set as 8. The mean tree size varied from 25 to 125, and in each of the four datasets, all the tree size values conformed to normal distribution with variance of 2. The results show that for the range queries, the percentages of accessed data with binary branch filtration were almost the same as the result size for various tree size values. Histogram filtration needed to access much more data to process the same queries on the same dataset. When the mean value of tree size was 125, the binary branch filtration outperformed histogram filtration by more than a factor of 70 for range queries. The reason is that with label number and fanout almost fixed, the height, degree and the label histograms could vary little. The histogram information blurs the distance identification. On the other hand, the increase of size led to the increase of the edit distances. So the larger size caused worse performance of both our algorithm and histogram filtration methods. However, binary branch filtration still outperformed histogram filtration for various tree size. As can be seen, when the mean values of the tree size increased, the time consumption for the computation of the real distances increased quadratically. So, although the result size was almost the same, the sequential search time was too long for the datasets with large size. Thus, our algorithm is quite efficient for the similarity search on the large trees.

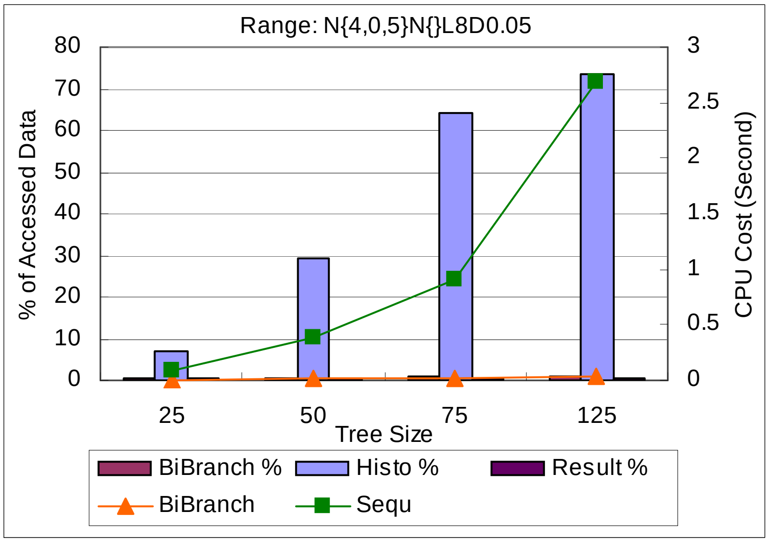


Figure 3.11: Sensitivity to Size of Trees for Range Queries

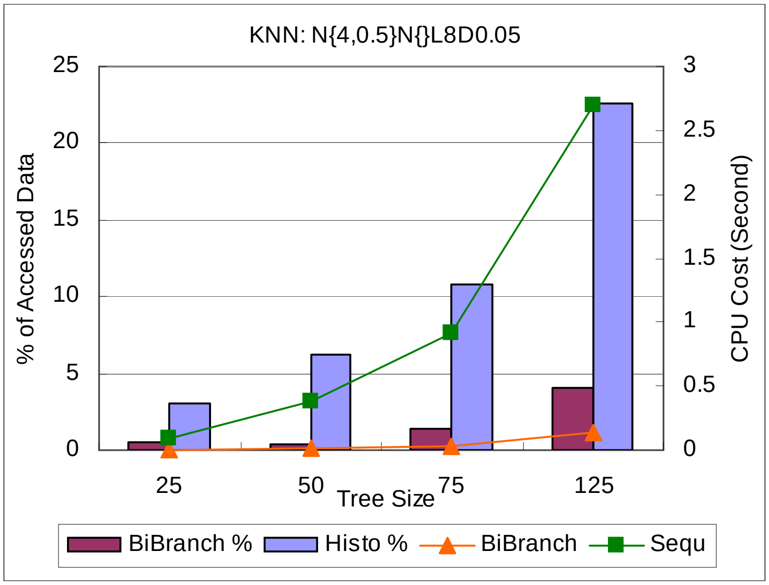


Figure 3.12: Sensitivity to Size of Trees for k-NN Queries

Figure 3.13 and Figure 3.14 show how the algorithms performed with the number of labels in the datasets increased. The parameters for the tree size and the fanout conformed to $N\{50, 2.0\}$ and $N\{4, 0.5\}$ respectively. The size of the label universals for the four datasets vary as 8, 16, 32, 64. As shown in the figures, the binary branch filtration algorithm always outperformed the histogram algorithm. When there were eight labels in the dataset, the performance of histogram filtration was less effective than binary branch filtration by more than a factor of 20. In the two figures , with the increase of the number of labels from 8 to 32, the histogram filtration improved much. The reason is that the label histogram can perform better with a large label size. However, since the histogram vector size was set to be comparable to the binary branch vector representation, and since the mean values of the distance increased with the label size becoming larger, the performance began to degrade when the number of labels was larger than 32 for both the range queries and k-NN queries.

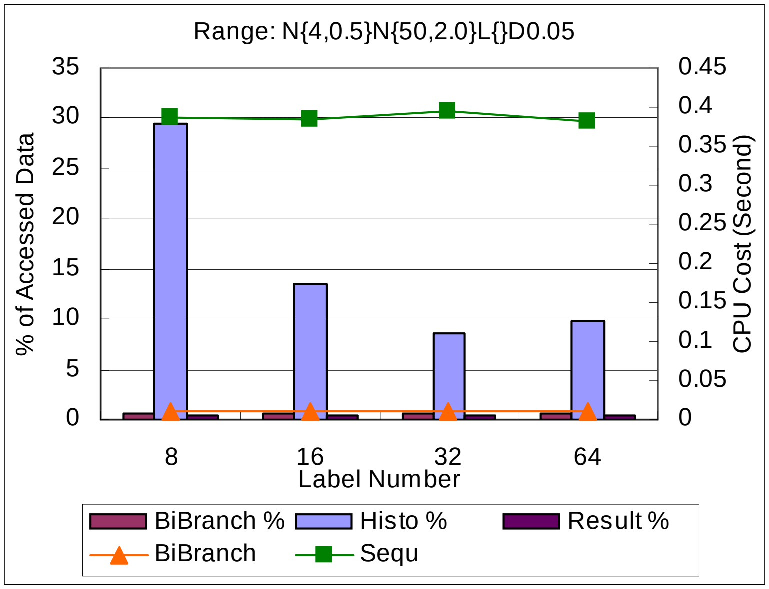


Figure 3.13: Sensitivity to Number of Labels in Trees for Range Queries

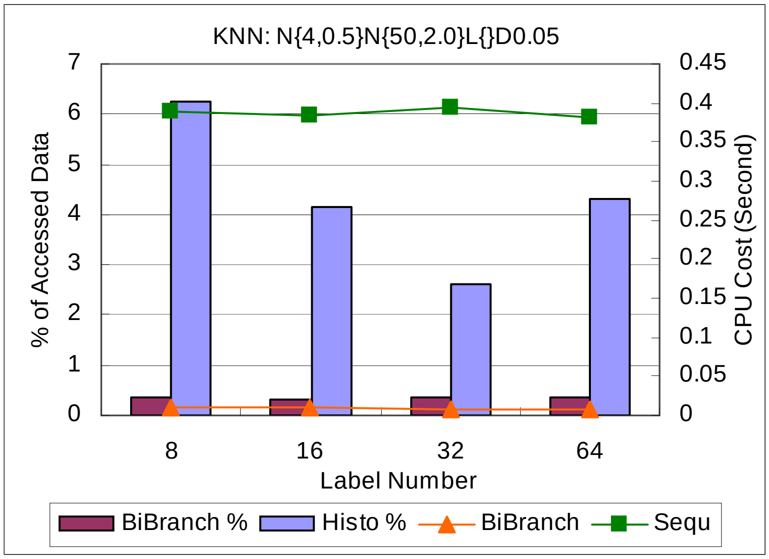


Figure 3.14: Sensitivity to Number of Labels in Trees for k-NN Queries

3.4.2 Similarity Query Performance

The experiments described in this part were conducted to compare the performance of the two filtration algorithms for the queries with different parameters. Figure 3.15 and Figure 3.16 show the performance of the two algorithms for k-NN queries and range queries on the $DBLP$ data. We randomly chose 2000 data objects from the whole $DBLP$ dataset. 100 queries were randomly chosen from this set. The average tree size of the the data was 10.15; And the average distance among the data was 5.031;

Figure 3.15 displays the k-NN query results on $DBLP$ data with the k varied from 5 to 20. The CPU time for sequential search is also plotted in the figure. It can be seen that the binary branch filtration accessed much less data than the histogram filtration. It performed one to three times better than the histogram filtration. Since the $DBLP$ data clustered very well, the percentage of the accessed data was small and the search time of binary branch filtration was only 1/6 of the sequential search time.

Figure 3.16 shows the results of range queries on $DBLP$. When the range remained less than the average distance among the data, the binary branch method clearly had better filtration power than the histogram method. As the range continued to increase to 10, the performance difference of the two methods decreased. The reason is that the result set was almost the whole dataset. Compared to the results of the percentage of data accessed in the previous experiments, the binary branch filtration here showed a smaller advantage over histogram filtration. This is due to the fact that the $DBLP$ data consists of shallow and small tree data, and the relatively small size of the binary branch universal set blurs the distinctions among data.

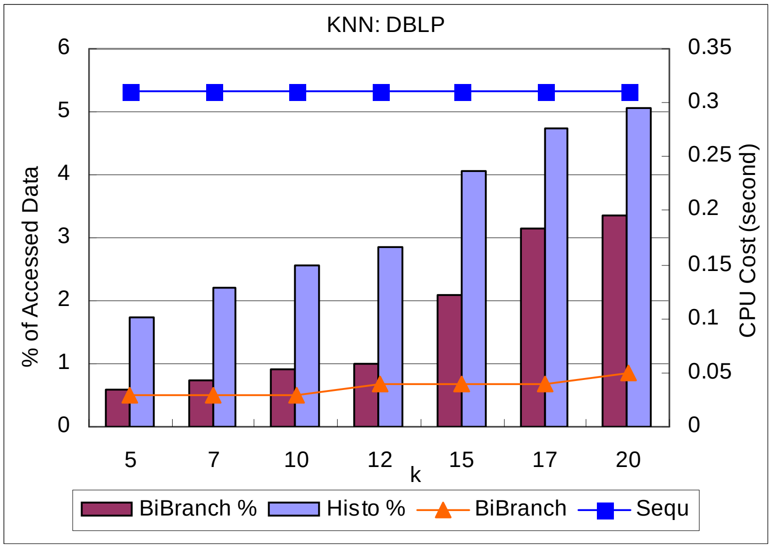


Figure 3.15: k-NN Searches on $DBLP$

3.4.3 Pruning Power With Respect To Binary Branch Levels

Figure 3.17 shows the distribution of data according to distances between the data and the queries on $DBLP$. The results here were averaged on the query number. The data distribution on three kinds of distance are plotted: edit distance, binary branch distance ($BiBranch(2)$ in Figure 3.17) and histogram distance between each data and query. Data distribution according to three and four-level binary branch distances ($BiBranch(3)$ and $BiBranch(4)$ in Figure 3.17) are also plotted. It can be seen that two-level binary

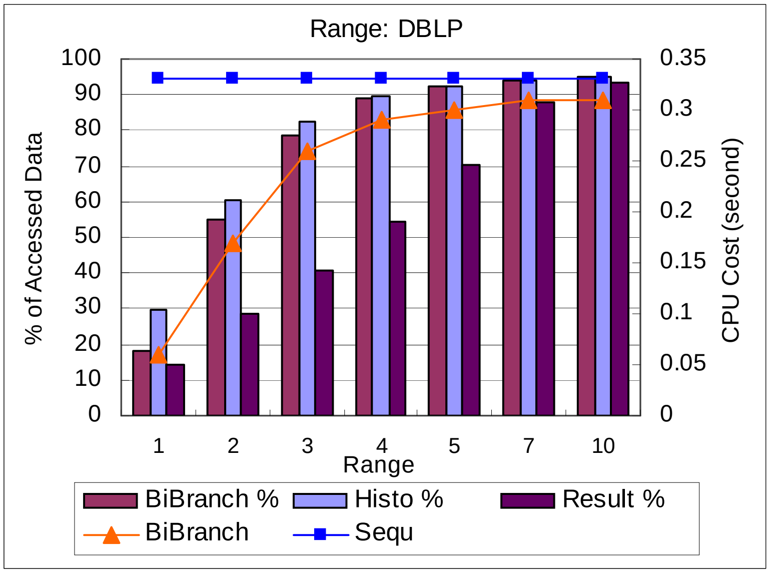


Figure 3.16: Range Searches on $DBLP$

branch distance is a better lower bound of edit distance than the histogram distance. Thus it can filter out much more data than histogram filtration when processing similarity search. When the distance is less than 3, three and four-level binary branch distance are also better than histogram distance. When the range is larger than 3, the data distribution is almost the same for three and four-level binary branch distance and histogram filtration distance. According to the definition of the multiple level binary branch, for the shallow tree-structured data like $DBLP$ records, multiple level binary branch distance is not an efficient lower bound for edit distance.

From the above analysis, it is obvious that the binary branch filtration is robust since it outperforms histogram filtration on processing various types of datasets and on various settings of the queries. It is particularly suitable for processing real datasets in spite of their skewed nature. This may be because it encodes structure information as well as the label information into the binary branch vector representations and positional sequences. In contrast, histogram filtration blurs the distinctions between trees since it uses only the histogram information, and the height, fanout and label histogram are considered

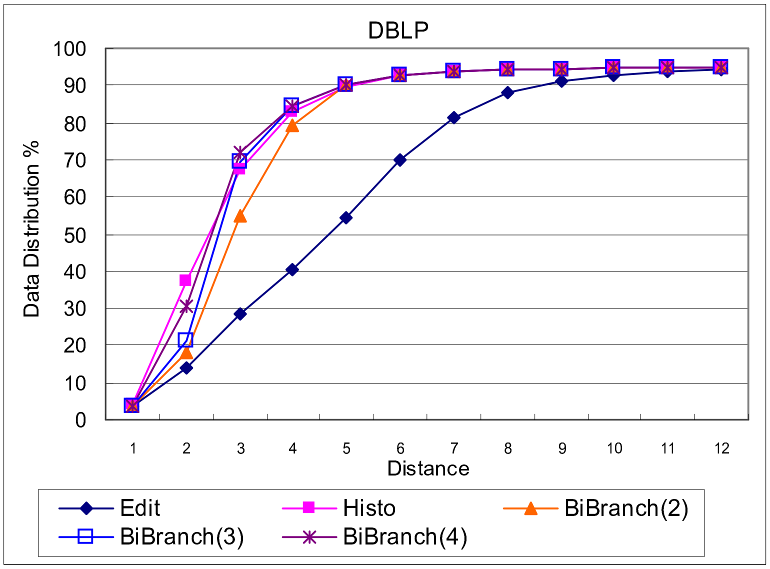


Figure 3.17: Data Distribution on Distance

separately.

3.5 Conclusion

XML data is becoming ubiquitous as it can express the hierarchical dependencies among data components and can be used to model data in many applications. Just as for other types of data, searches based on similarity measure are in the core of many operations for tree-structured data. However, the computational complexity of the general dissimilarity measure (i.e., the tree-edit distance) render the brute force methods prohibitive for processing large trees in huge datasets.

In this chapter, an efficient method based on the binary tree representation is proposed. The XML data tree is transformed into binary branch numerical vectors. This *characteristic vector* records the structural information of the original tree, and the L_1 norm distance on the vector space is proved to be the lower bound of the tree-edit distance. Moreover, the vector representation of trees can be generalized by using multiple level binary branches; this enables the structural information to be encoded in different granularity. Since the novel lower bound is much easier to obtain than the original distance measure, it can be embedded in the filter-and-refine architecture to reduce the computation of real edit distance between data and queries and guarantee no false negatives. In addition, novel filter-and-refine similarity search algorithms are given, which exploits the positional binary branch properties to obtain a better lower bound of edit distance. The results of the experiments show that the new algorithm is robust to varying dataset features and query parameters. The pruning power of the new algorithms leads to both CPU and I/O efficient solutions.

Chapter 4

Accelerating XML Twig Pattern Matching

4.1 Introduction

As business and enterprizes generate and exchange XML data more often, there is an increasing need for efficient processing of pattern queries on this type of data. Searching for all occurrences of a twig pattern in the XML database is a core operation in XML query processing. An XML twig query, represented as a labeled tree, is essentially a complex selection predicate on both *structure* and *content* of the XML documents. While value-based conditions can be efficiently evaluated with traditional indexing schemes, answering the structural constraints is a challenging task. This chapter is mainly focused on twig queries which are the basic component of declarative XML query languages, such as XQuery and XPath.

The previously proposed methods [69, 20, 73, 28, 74, 55] have been proved to be I/O optimal only to some specific query classes. The problem of the binary structure join methods is mainly due to the query decomposition [69]. While for holistic twig join methods, the problem is caused by the sequential scan of the element streams. At any point, only the cursor elements and the elements stored in the stacks are visible. However, according to these information, it is impossible to completely identify whether the cursor elements are in a match to the whole twig pattern. Thus some “useless” partial solutions which do not contribute to the final answers have to be output to avoid false dismissal.

Another inspiring observation is that all the previous holistic approaches solve the problem by producing the matching bindings for ***all*** nodes in a twig query. However, in a practical application, this requirement is not necessary. In this thesis, query nodes whose matches should all be retrieved are referred to as *distinguished* nodes, and those used only for qualifying the structural relationships of a query are referred as *existential* nodes. As mentioned in Chapter 1, straightly utilizing the results of previously proposed methods and do *projection* on those distinguished nodes matches is not efficient. Firstly, such method outputs all matches of existential nodes and is not I/O optimal; Secondly, even if only matching elements for distinguished nodes are considered, prior algorithms still show the non-optimality by outputting many matches of distinguished nodes that do not belong to final answers.

In this chapter, theoretical analysis of the reasons for the non-optimality of the previous methods is given. And the practical requirements for answering twig queries is exploited to develop two novel twig matching algorithms which do not output the intermediate path matching results. By utilizing a limited size of main memory, these algorithms are guaranteed to be optimal for a much broader class of queries than the prior methods. The rest of the chapter is organized as follows. In Section 4.2, the definition of bounded and unbounded matching blocks is given. I introduce as well a set of theory to expose the relationships between query structures and optimal holistic join algorithms in this part. Section 4.3 and Section 4.4 present two new holistic twig join algorithms based on the *containment* and *prefix* numbering schemes respectively, together with the correctness and the complexity discussion of them in Section 4.5. Section 4.6 presents comprehensive experimental studies on the performance comparison between the novel algorithms proposed in this chapter and the prior methods, as well as the comparison between the two new algorithms. Section 4.7 concludes the chapter.

4.2 Theoretical Analysis

In this section, I theoretically analyze the reason for the non-optimality of the previous holistic twig join algorithms and the possibility to design new holistic algorithms that are optimal for a larger query class than the previous methods.

In this chapter, the pattern queries are referred to by Q, and the nodes in it are denoted by q (with its subscript i representing the ith node according to the preorder traversal of it). The XML dataset are denoted by D (with its subscript i represent the ith XML data). As in the previous literatures of holistic approaches, a structure named XML element stream is associated to each query node. The stream is a posting list (or inverted list) containing the encoding numbers of the XML elements which have the same label, and all elements are ordered according to the *document positions*. More specifically, for the *containment* numbering scheme, all elements are sorted by the value of the pair $(DocId, LeftPos)$; while for the *prefix* numbering scheme, all elements are sorted by the lexicography order. There is a unique cursor for each stream. It moves in the single direction to scan all elements once in increasing order. The element pointed by the cursor in a stream is referred to as cursor element. The stream of query node q_i is denoted as S_{q_i} and the elements in it is denoted as e_{q_i} (or with the prime characters).

4.2.1 Matching Block

The existing holistic twig join algorithms consists of two phases: (i) in the first phase, the partial solutions to each individual root-to-leaf path expression are output as as intermediate results; and (ii) in the second phase, the element paths are merged to produce the final answers for the whole twig query. However, for queries with PC relationships, many state-of-the-art algorithms cannot guarantee that each intermediate solution output in the first phase can be merged with other partial solutions in the second phase. In other words, many useless intermediate solutions may be produced in the first phase, as shown in the following example.

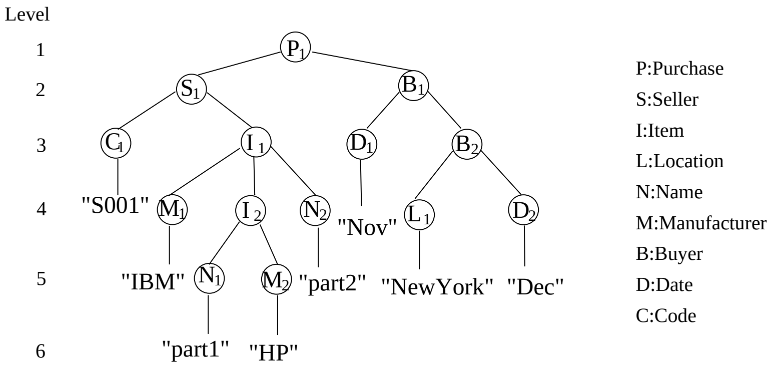


Figure 4.1: A sample XML tree

Example 4.2.1. Consider the document in Figure 4.1 and the query $I^*[M]/N$. Firstly, I_1, M_1 and N_1 are scanned. we cannot determine whether or not I_1 is a query answer. At this point, it is to know that I_1 has a child M, i.e., M_1. However, N_1 is not child of I_1. We do not know whether I_1 has a child N after N_1. At the same time, N_1 may have parent M after M_1 (In this example, N_1 has M_2 as its parent). Now holistic algorithms meet a dilemma, i.e., no stream can be advanced before we determine whether I_1, M_1 or N_1 is in an answer. Previous methods hastily pushing I_1 into stack and output the path (I_1, M_1), which may become useless intermediate path if there were not N_2 in join data. □

In the following , we formalize the observation in Example 4.2.1 into a concept, **matching block**, which describe a situation wherein, in order to guarantee the optimality of algorithm, two or more different data streams have to wait for the other to advance elements, so neither ever does.

Definition 4.2.2 (Matching Block). Given an XML document D and a query Q, assume that q_i, q_j are two query nodes in Q. Let e_{q_i}, e'_{q_i} (in order) be two elements in data stream S_{q_i}. Similarly, let e_{q_j}, e'_{q_j} (in order) be two elements in data stream S_{q_j}. We say that the 4-tuple $< e_{q_i}, e'_{q_i}, e_{q_j}, e'_{q_j} >$ is a *matching block* for Q on D if and only if the pairs of elements (e_{q_i}, e'_{q_j}), (e'_{q_i}, e_{q_j}) are the matching bindings to Q, but (e_{q_i}, e_{q_j}) and (e'_{q_i}, e'_{q_j}) are not. (Figure 4.2 illustrates this concept graphically) □

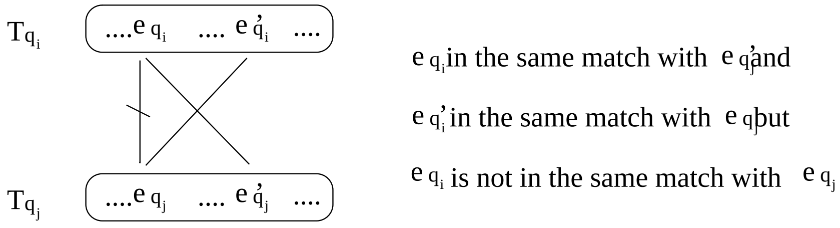

Tq_i
....e q_i .... e q'_i ....
e q_i in the same match with e q'_j and
e q'_i in the same match with e q_j but
e q_i is not in the same match with e q_j
Tq_j
....e q_j .... e q'_j ....

Figure 4.2: Illustration to Matching Block

In Example 4.2.1, $< M_1, M_2, N_1, N_2 >$ is an instance of matching block since (M_1, N_2), (M_2, N_1) are components of the matching tuples (I_1, M_1, N_2) and (I_2, M_2, N_1) but (M_1, N_1) and (M_2, N_2) are not. The possibility of the existence of *matching blocks* forces holistic algorithms to store I_1 and M_1 in the stacks to avoid the loss of results. However, they may not participate in the whole matches of the query. So, “useless” intermediate path solutions may be output and thus causes the sub-optimality. The detailed analysis is given at Chapter 2 Section 2.4.3. The following lemma identifies a query class where we cannot find any document with blocks.

Lemma 4.2.3. *Suppose Q is a twig query with only AD relationships in all structural predicates, there exists no matching block for Q on any document D.*

Proof. Let q_i and q_j be any two query nodes in Q. We prove this by rule of contradiction. Assume that an instance of matching block $< e_{q_i}, e'_{q_i}, e_{q_j}, e'_{q_j} >$ occurs when evaluating Q on some document D. Without loss of generality, let e_{q_i} precede e_{q_j} according to the preorder traverse of the data tree T. Then, there can be two cases:

1. q_i is an ancestor of q_j in Q. Obviously, e_{q_i} should be an ancestor of e'_{q_i}. Otherwise, e_{q_i} must end before the start of e'_{q_i}, and thus the start of e_{q_j} and e'_{q_j}. Then e_{q_i} and e'_{q_j} cannot satisfy AD relationship, which is contrary to the definition of matching block. Because e_{q_i} is an ancestor of e'_{q_i}, e_{q_i} is also an ancestor of e_{q_j}. Therefore, $< e_{q_i}, e_{q_j} >$ is also a matching binding, which contradict the definition of block.

2. q_i and q_j are in the different root-to-leaf pathes in Q. Assume that in Q, node q_h is the lowest common ancestor (abbr. LCA, i.e., the lowest node in the twig which is the ancestor of both q_i and q_j). In these two matches, if q_h binds to one element e_{q_h} in the data, then e_{q_h} and e_{q_i} must match the path query between q_h and q_i; e_{q_h} and e_{q_j} must match the path query between q_h and q_j. So e_{q_h}, e_{q_i} and e_{q_j} should be in one match. This is contrary to the fact that $< e_{q_i}, e'_{q_i}, e_{q_j}, e'_{q_j} >$ is a block.

   Otherwise, assume q_h matches to two different nodes e_{q_h} and e'_{q_h} (in order) in these two matches. Similar to the analysis of Case 1, e_{q_h} should be the ancestor of e'_{q_h}. And e_{q_h}, e_{q_i} and e'_{q_j} is in one match; e'_{q_h}, e'_{q_i}, and e_{q_j} is in another. Since there are only AD edges between q_h, q_i and q_h, q_j, then e_{q_h} must also be the ancestor of e'_{q_h} and e_{q_j}. So e_{q_h}, e_{q_i} and e_{q_j} are also in a matches which is contradict the block assumption. The matches among e'_{q_h}, e_{q_i} and e'_{q_j} are similar.

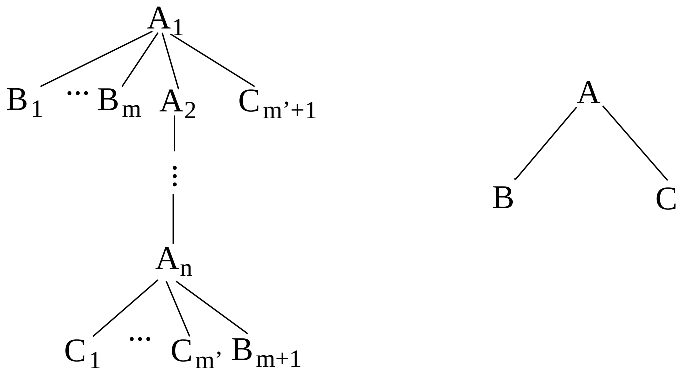
□

From the above reasoning, we know that for two query nodes q_i and q_j $(i < j)$ with block matches in their streams, if they are in one query path, then there should be at least 1 PC edge between them and the two blocked matches of q_i must be in the same data path. If q_i is not an ancestor of q_j, the two matches of their *LCA* query node q_h, e_{q_h} and e'_{q_h} should be in one path. According to Lemma 4.2.3, no block can occur during evaluating queries with only AD relationships. Thus all the holistic join algorithms can guarantee the optimality for such queries.

However, blocks do not necessarily lead to the non-optimality of holistic algorithms. Next, we define one type of blocks which can be processed optimally by caching limited number of elements in the main memory.

Definition 4.2.4 (Bounded and Unbounded Matching Block). Given a query Q and an XML document D, assume that $< e_{q_i}, e'''_{q_i}, e_{q_j}, e'''_{q_j} >$ is an instance of matching block for Q on D. $< e'_{q_i}, e''_{q_i}, e'_{q_j}, e''_{q_j} >$ (e_{q_i}, e'_{q_i}, e''_{q_i}, e'''_{q_i} and e_{q_j}, e'_{q_j}, e''_{q_j}, e'''_{q_j} are in order respectively) is an **embedded block** in $< e_{q_i}, e'''_{q_i}, e_{q_j}, e'''_{q_j} >$ if $< e'_{q_i}, e''_{q_i}, e'_{q_j}, e''_{q_j} >$ is also a matching block.

Furthermore, if the number of distinct elements that are involve in some embedded blocks between e_{q_i} and e'''_{q_i} in S_{q_i} (or between e_{q_j} and e'''_{q_j} in S_{q_j}) is no more than the maximum depth of document D, then $< e_{q_i}, e'''_{q_i}, e_{q_j}, e'''_{q_j} >$ is called a **bounded matching block**(BMB), otherwise it is an **unbounded matching block** (UMB). □



Document Query

Figure 4.3: Example of BMB and UMB

For example, consider the query and the document in Figure 4.3. $< A_1, A_n, C_1, C_{m'+1} >$ is a bounded block, because the number of distinct elements between A_1 and A_n that belong to some embedded block is no more than n, which is bounded by the depth of the document. In contrast, $< B_1, B_{m+1}, C_1, C_{m'+1} >$ is an unbounded block. This is because m or m' is not bounded by the depth of the document and the number of distinct elements between B_1 and B_{m+1} (or between C_1 and $C_{m'+1}$) that are involved in the embedded blocks may be much greater than the depth of documents.

Lemma 4.2.5. *UMB can only occur between query nodes which are in different query path.*

Proof. Assume that q_i is an ancestor query node of q_j, and there is a block match $< e_{q_i}, e'_{q_i}, e_{q_j}, e'_{q_j} >$ between them. It is similar to the 1st case of Lemma 4.2.3. Elements e_{q_i} and e'_{q_i} should be in one path, otherwise e_{q_i} and e'_{q_j} cannot be in the same path and cannot in a match. Thus the lemma is proved. □

Lemma 4.2.6. *Suppose Q is a twig query with only AD relationships to connect branching nodes, given any document D, there cannot be any unbounded matching block when evaluating Q on D.*

Proof. According to Lemma 4.2.5, UMB only occurs between the query nodes which are in the different path. Let q_i and q_j are two query nodes in different path of Q and let q_h be their LCA. Assume there is an $UMB < e_{q_i}, e'_{q_i}, e_{q_j}, e'_{q_j} >$ on them and e_{q_i} precedes e_{q_j} in document order. Similar to the 2nd case of Lemma 4.2.3, q_h must have two different matches e_{q_h} and e'_{q_h} (in order). Meanwhile, e_{q_h} must be an ancestor of e'_{q_h}. Since there are only AD relationship predicates under q_h, e_{q_h}, e_{q_i} and e_{q_j} must be in one match. So is e'_{q_h}, e'_{q_i} and e'_{q_j}. Thus, there cannot exist any matching block at all between q_i, q_j and q_h.

□

Since the number of distinct elements involved in BMB is less than the depth of the documents, it is reasonable to assume that holistic algorithms can cache all these elements in the main memory. However, for UMB, we *cannot* assume that holistic algorithms can cache all elements involved in the main memory. In the following section, a query class is identified where there is only BMB on any given document. Thus, this query class can also be processed *optimally* by holistic algorithms. The difference between BMB and UMB motivates the design of new query processing algorithm.

4.2.2 Enlargement of the Optimal Query Class

Lemma 4.2.6 identifies a query class that only causes BMB. The analysis of this section shows that this optimal query class can be substantially enlarged if the difference between the *distinguished* nodes and the *existential* nodes is exploited. As can be seen in Example 4.2.1, the *distinguished* node in the query is only I (not M or N). $< M_1, M_2, N_1, N_2 >$ may become an unbounded block if there are many "M" elements being I_1's children before N_2 and many "N" elements being I_2's children before M_2. However, this unbounded block can still be efficiently processed. Instead of outputting the concrete path $< I_1, M_1 >$, only the information that I_1 has an appropriate child M (M_1 here) need to be maintained. And only N_2 can fulfill the matching condition and trigger the output of I_1. Thus the streams can be advanced without loss the results.

The above observation shows that the existence of unbounded block in the undistinguished (i.e. *existential*) data streams can be conquered by recording the matching information in the main memory. In the rest of this section, theorems are developed to identify the query class on which all unbounded blocks only occur in undistinguished data streams. To achieve this purpose, the definition of the *distinguished path* and *optimal distinguished node* in the query tree is given as follows:

Definition 4.2.7 (Distinguished Path). Assume the query node q_d in Q is the distinguished node, the query path from root to q_d consists of the distinguished path

Definition 4.2.8 (Optimal Distinguished Node). A query node q_i in Q is optimal if and only if

- q_i is the root of Q or,
- the parent node $q_{parent(i)}$ of q_i is optimal and all the other child nodes of $q_{parent(i)}$ must connect to it through AD relationship

Suppose that Q is a twig query with the distinguished node q_d. q_d is called optimal distinguished node if it is optimal.

Theorem 4.2.9. *Assume that Q is a twig query with a single distinguished node q_d. If it is an optimal distinguished node, then there is no UMB involving the stream S_{q_d} on any document D for Q.*

Proof. As shown in the Lemma 4.2.6, for the query nodes in one path, there cannot be unbounded matching blocks between elements in their streams. Since all the descendant query nodes of the distinguished node q_d is in a path with it, we know that all the query nodes under q_d cannot have UMBs with it. And all the nodes on the distinguished path above it cannot have UMB with the distinguished node. The proof of the other nodes are given here. There are two cases for each of the branching node associated with the distinguished path. One case is that the edge of the distinguished path under the branching node is PC relationship constrain. Another case is that the edge of the distinguished path under the branching node is AD predicate. Figure 4.4.(a), (b) shows these two cases respectively, where q_a is the branching node under study and q_c represents its other child nodes. In Figure 4.4 the dashed lines represent the elliptical paths and the dots represents the elliptical parts of the query.

1. For the first case shown in Figure 4.4.(a), assume that there is matching block $< e_{q_d}, e'_{q_d}, e_{q_j}, e'_{q_j} >$ between q_d and q_j. We know that in these two matches, node q_a must be matched to two nodes e_{q_a} and e'_{q_a} in order. Otherwise, e_{q_d} and e_{q_j} (e'_{q_d} and e'_{q_j}) are actually in one match, which is contrary to the definition of matching block. And it is sure that e_{q_a} and e'_{q_a} are in one data path. Otherwise, e'_{q_j} starts

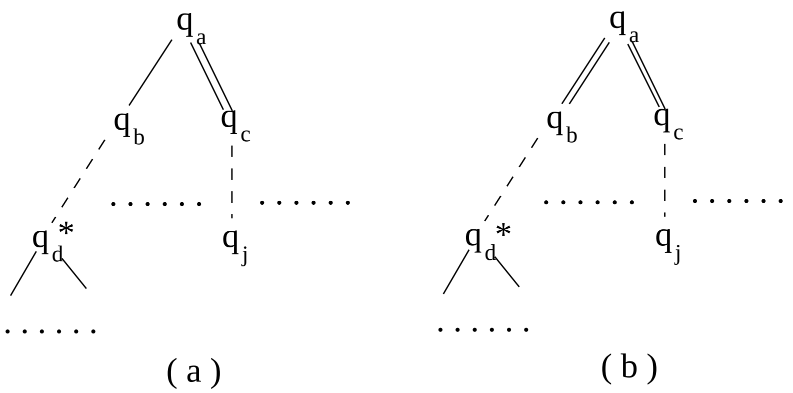


Figure 4.4: Illustration of Theorem 4.2.9

after e_{q_a} ends and they cannot be in the same match. Because the predicates are AD relationship between q_a and q_c, e_{q_a} and e_{q_j} should also match the path query between q_a and q_j. Thus e_{q_a}, e_{q_i} and e_{q_j} can be in one match, which is contrary to the assumption that $< e_{q_d}, e'_{q_d}, e_{q_j}, e'_{q_j} >$ is a matching block. Thus there cannot be UMB for any document. The proof for the case $< e_{q_j}, e'_{q_j}, e_{q_d}, e'_{q_d} >$ is similar.

2. For the second case shown in Figure 4.4.(b), the proof is similar to Lemma 4.2.5.

□

Theorem 4.2.9 shows that the optimal query class can be much larger than the previous ones. An excellent example is that when the query *root* node is the single *distinguished* node, the optimality of the new holistic algorithms can be guaranteed regardless of the PC and AD combinations under it.

Theorem 4.2.10. *If Q is a tree pattern query with multiple distinguished node and all of them are optimal, then there cannot be UMB involving the matches of any distinguished query node.*

Proof. Theorem 4.2.10 is a natural extension of Theorem 4.2.9. We only need to prove the case between any two distinguished nodes q_i and q_j in the query Q. Obviously, if q_i and q_j are in the same query path, then there is no UMB between them.

Assume that q_i and q_j are at different path and the LCA of q_i and q_j is q_a. From Theorem 4.2.9, we know that under node q_a, there can exists at most one PC predicate among the constrains connecting the two child nodes of q_a which are on the two distinguished path of q_i and q_j respectively. Similar to the proof of the first case of Theorem 4.2.9, we know that there cannot be matching block between q_i and q_j. □

Figure 4.5 show some examples of the newly extended optimal queries. It should be noted that these theorems are not associated with any specific labeling scheme. In the next section, two novel algorithms are developed, which are based on the two popular labeling schemes (i.e. *containment* and *prefix* schemes) respectively which are optimal to query class specified by the above theorems.

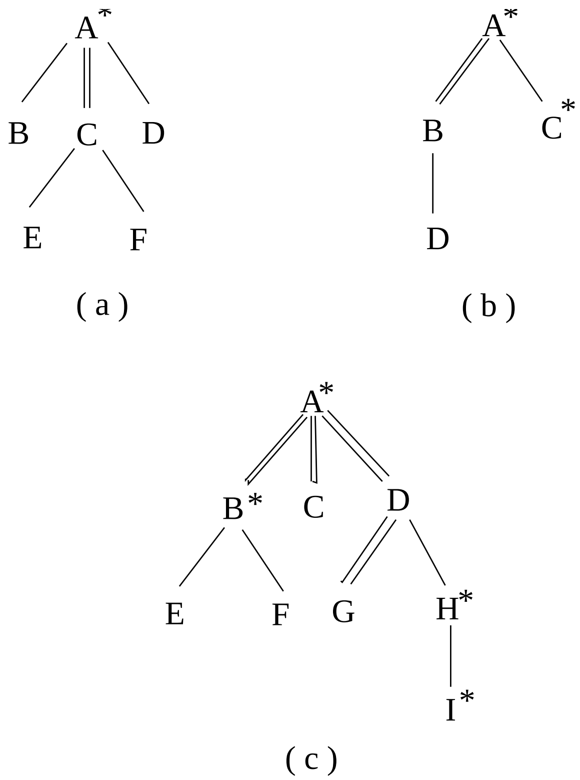


Figure 4.5: Optimal query nodes

Remark 4.2.11. The non-optimality of holistic twig algorithms originates from the possible existence of *matching blocks* in data streams. When there is any block in data streams, TwigStack[20] may show its non-optimality by outputting “useless” intermediate results. But our above analysis suggests that not all blocks undoubtedly lead to the non-optimality for holistic algorithms. In particular, blocks can be categorized to two types: BMB and UMB, wherein BMB can be conquered by caching limited number of element in main memory. As an example, previous algorithm TwigStackList[73] efficiently handles BMB and guarantees the optimality for queries which have PC edges in non-branching edges. Unfortunately, TwigStackList cannot be extended to handle UMB efficiently, because that requires to cache too many elements in the main memory. In the worst case, all elements in a document should be cached in the main memory. However, according to the above analysis, the UMB in *undistinguished* data streams can still be efficiently processed by recording some matching information and by selectively storing limited elements in main memory. Since previous algorithms do not differentiate *existential* nodes from *distinguished* nodes, they cannot explore this improvement space.

Algorithm iTwigJoin proposed by Chen et al [28] can identify a larger optimal query class than TwigStack and TwigStackList since, in essence, iTwigJoin solves the *matching block* by separating elements to different streams. Thus, our theory is applicable to their work for making further improvement.

In the following section, two novel algorithms are proposed to evaluate XML twig queries. The challenge is to implement the theoretical results to enlarge the optimal query class. As an evidence of the *generality* of the theoretical results, two algorithms are proposed, which are based on the popular *containment* and *prefix* numbering schemes respectively.

4.3 TwigContainment

TwigContainment, inspired by TwigStack [20], is based on containment numbering scheme. Firstly, the algorithm for queries with a single distinguished node is presented. And then it is naturally extended to support multiple *distinguished* nodes. The section begins with the introduction of the data structures and notations.

4.3.1 Data Structure

The query twig and the streams of query nodes are modeled similarly as in the previous work of TwigStack [20]. A twig query on XML can be represented with a small tree structure. There are four self-explaining functions of the twig node: $isRoot(q)$ and $isLeaf(q)$ verify whether q is the root node or a leaf node. $isDist(q)$ and $isAnceDist(q)$ verify whether q is distinguished node and whether q is ancestor of the distinguished node respectively. $parent(q)$, $children(q)$ and $subtreeNodes(q)$ retrieve the parent node of q, the child nodes of q and the nodes in the subtree rooted at q respectively. And functions $PC(q_i, q_j)$ ($AD(q_i, q_j)$) is used to check whether q_i is the parent (resp. ancestor) node of q_j.

There is a data stream S_q associated with each query node q, in which all the elements can satisfy the predicate specified by q. The record of each element in S_q consists of its positional representation $(DocId, LeftPos : RightPos, Level)$, where $DocId$ is the data record id, $LeftPos$ and $RightPos$ are its containment numbers and $level$ records on which level the element is in the data. We use e_q to refer to these elements. [1] $Next(S_q)$ denotes the cursor element of S_q. And $nextL(S_q)$ can retrieve the $leftPos$ value of the cursor element. The stream can be advanced to the next element in S_q with the procedure $advance(S_q)$. Each stream is supplemented by an virtual ending element represented by

[1] The description of the algorithm ignore $DocId$ firstly. However, it is easy to extend it to deal with $DocId$.

(∞, ∞, ∞). And the end of the stream can be checked by function $eof(S_q)$. Assume q_c is a child node of q, $Rel_d(e_q, e_{q_c})$ check if elements e_q and e_{q_c} satisfy the relationship between q and q_c. If it is AD predicate between them, this is implemented by checking whether $e_q.LeftPos < e_{q_c}.LeftPos$ and $e_q.RightPos > e_{q_c}.RightPos$. For the PC relationship, other than the above requirement, $e_q.Level$ should equal to $e_{q_c}.Level - 1$ in order that they satisfy PC relationship.

The stack structure in TwigStack is extended in the algorithm here to present matching results. In particular, there is an extended stack ES_q associated with each query node q. Each item in stacks consists of a 4-tuple $(num(e_q), bitVector, outputList, ptrP)$. $num(e_q)$ is the encoding number of the corresponding element e_q from S_q. The length of $bitVector$ equals to $|children(q)| + 1$. The first $|children(q)|$ bits are matching bits. If the element in an stack item has the correct extension of the child node q_c, then its corresponding bit is “1”. Otherwise, it is reset. The last bit is a flag to identify whether the item of e_q is referred to by a $ptrP$ in child q_c's stack if q_c is in distinguished path. And $outputList$ contains the elements which match the distinguished nodes that possibly become the final query answers. $prtP$ points to an item in $ES_{parent(q)}$ with which e_q satisfy the relationship constrains between q and $parent(q)$. There are several functions on the stacks. $empty(ES_q)$, $pop(ES_q)$, $push(ES_q, e_q, 0, NIL)$, $topL(ES_q)$, and $topR(ES_q)$. The last two operations return the $leftPos$ and $RightPos$ attribute of the element in top item. $push(ES_q, e_q, 0, NIL)$ is used to push the new item into the stack, with e_q, 0, and NIL as the value of the first three fields. Meanwhile, $ptrP$ is pointed to the top of $ES_{parent(q)}$. Although the items of the stack can only be pushed into or popped up from the top, all the items can be visited during processing, which is implemented by the operations on the items of the extended stack. $bitVec(ES_q, Itm)$, $outputL(ES_q, Itm)$, $elem(ES_q, Itm)$ and $prt(ES_q, Itm)$ are used to retrieve the four fields of the Itmth item in ES_q respectively. $bit(ES_q, q_c, Itm)$ retrieve the q_c bit of the Itmth item in stack ES_q.

$bit(ES_q, q_c, Itm).set$ is to change the bit from “0” to “1”.

Given these differences of the stack definition, it is still used to record the partial results of the query. Similar to the stacks in TwigStack method, the elements in the extended stack from the bottom to the top satisfy the AD relationship. At every point during the computing, for each item in stack ES_q, (i) if all matching bits in $bitVector$ are “1”, then its element e_q is guaranteed to match the subtree query rooted with q. Therefore, if q is the root, then e_q is guaranteed to be the root of a match to the whole query, and (ii) $\forall e \ \in outputList$ is the query answer if and only if e_q match the whole twig query. Therefore, whether an element $e \ \in outputList$ is a query answer can be *accurately* described by the corresponding $bitVector$. For ease of description, the element $e_{q'}$ which matches to one of the other query nodes and satisfy the pattern between q and q' with e_q is defined as the *correlative node* of e_q, denoted as $corr(e_q, q')$. $corr(e_q, q')$ is not one and only.

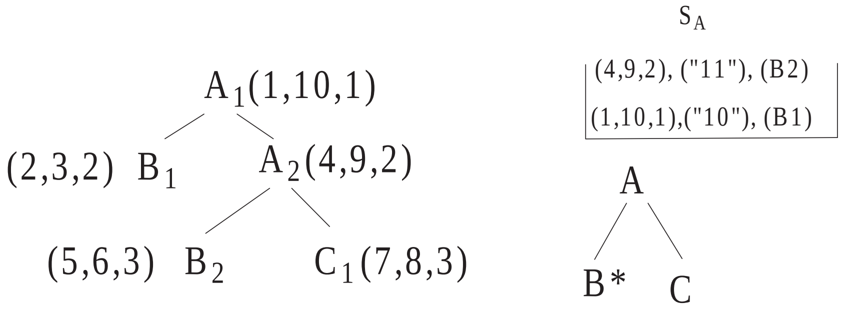


Figure 4.6: Stack Encoding of Query Results

Example 4.3.1. Figure 4.6 illustrates the stack configuration to node A in a twig query for a sample document. There are two items, corresponding to elements A_1 and A_2 in the stack S_A. Since A_1 has one child B_1 and no child element to match C, $bitVector$=“10”. In contrast, in the item for A_2, matching bits of $bitVector$ = “11″, because A_2 has two child B_2 and C_1, which satisfy the PC relationships in the query. Consequently, B_2 is the query answer. On the contrary, B_1 is not an answer.

Algorithm 3 TwigContainment

```
Input: $Q$ is a query twig pattern with distinguished node $q_d$
1: while $\neg(end(root(Q)))$ do
2:     $q_{act} = getMinSource(root(Q))$;
3:     $cleanStack(root(Q), nextL(S_{q_{act}}))$;
4:     if $isRoot(q_{act})$ or $\neg empty(ES_{parent(q_{act})})$ then
5:         $FLAG = moveStreamToStack(q_{act}, S_{q_{act}}, ES_{q_{act}})$;
6:     $advance(S_q)$;
7:     if ($isLeaf(q_{act})$ and $FLAG = true$) then
8:         $updateBit(q_{act})$;
Function: $end(q)$
  return $\forall q_i \in subtreeNodes(q) : isLeaf(q_i) \Rightarrow eof(S_{q_i})$;
Function $getMinSource(q)$
  return $q_i \in subtreeNodes(q)$ s.t. $nextL(S_{q_i})$ is minimal;
```

4.3.2 Algorithm

The main procedure of TwigContainment is depicted in Algorithm 3. Unlike TwigStack, this method operates in one phases. And merge-join part of different distinguished nodes' matches does not need a separate phase. The key idea is to repeatedly insert elements that are possible query answers into the $outputList$ of the extended stack of the *distinguished* node and propagate these elements up to the $outputList$ of the query root; the whole query is matched bottom up. Thus, the process is reverse to that of TwigStack. Firstly, I will give the processing algorithm for queries with 1 distinguished node. The extension to multiple distinguished nodes will be introduced later.

In Algorithm 3, the elements in the data streams of each query node are iterated till all the streams reach the ends. Line 2 identifies the stream containing the next node to be processed. That is the one whose cursor element is with the most small $LeftPos$ attribute. This guarantee that before an elements e_q is pushed into its stack ES_q, the elements $corr(e_q, parent(q))$ are already in $ES_{parent(q)}$.

In Line 3, $cleanStack()$ makes sure that before a element is pushed into its stack, all the elements in the stacks which end before it are recursively popped up from the stacks.

```
Procedure $cleanStack(q, nextL)$
for $\forall q_c \in children(q)$ do
    $cleanStack(q_c, nextL)$;
while $topR(ES_q) < nextL$ do
    if $isRoot(q)$ then
        if all matching bits of $bitVec(ES_q, top)$ are “1” then
            output $outputL(ES_q, top)$;
        else
            $Itm = nextMatch(q)$;
            Append $outputL(ES_q, Itm)$ with $outputL(ES_q, top)$;
    else if $isDist(q)$ then
        if $isLeaf(q)$ or all matching bits of $bitVec(ES_q, top)$ are “1” then
            Append $outputL(ES_{parent(q)}, ptr(ES_q, top))$ with $elem(ES_q, top)$ ;
    else if $isAnceDist(q)$ then
        if all matching bits of $bitVec(ES_q, top)$ are “1” then
            Append $outputL(ES_{parent(q)}, ptr(ES_q, top))$ with $outputL(ES_q, top)$ ;
        else
            $Itm = nextMatch(ES_q)$;
            Append $outputL(ES_q, Itm)$ with $outputL(ES_q, top)$;
    if $q$ is in distinguished path then
        $bit(ES_{parent(q)}, 0, ptr(ES_q, top)).reset$;
    $pop(ES_q)$;
```

```
Procedure $nextMatch(q)$
if AD not exists between $q$ and $q_d$ then
    return 0;
for $Itm_1$ from top to bottom do
    if $bit(ES_q, q_c, Itm_1)$ is “1”, with $q_c$ under $q$ in distinguished path then
        break;
return $Itm_1$;
```

The details are shown in Procedure $cleanStack()$. It has 3 functionalities. Firstly, for those elements which have the descendant extension, but matches to existential nodes, it only maintains their matching information and pops out them from stack. Secondly, for those elements which have descendant extension and match to distinguished node, it merges the matches of the distinguished nodes in their $outputList$ to that of the correlative element in the parent stack, and then pops them out from the stack. Thirdly, it is used to popped out and skip the elements which do not have descendant extension.

Procedure $updateBit(q)$

```
1: if $isLeaf(q)$ then
2:     if $PC(parent(q), q)$ then
3:         $bit(ES_{parent(q)}, q, top).set$;
4:     else
5:         for $\forall Itm \in ES_{parent(q)}$ do
6:             $bit(ES_{parent(q)}, q, Itm).set$;
7:     $updataBit(parent(q))$;
8: else
9:     $FLAG = 0$;
10:    for $\forall Itm_1 \in ES_q$ do
11:        if all matchhing bits $bitVec(ES_q, Itm_1)$ arel "1" then
12:            for $Itm_2$ from $ptr(ES_q, Itm_1)$ down to 0 do
13:                if $bit(ES_{parent(q)}, q, Itm_2)$ is "0" then
14:                    $bit(ES_{parent(q)}, q, Itm_2).set$;
15:                else
16:                    $FLAG = 1$;
17:                    break;
18:                if $PC(parent(q), q)$ then
19:                    break;
20:            if $FLAG == 1$ then
21:                $FLAG = 0$
22:                break;
23: if $\neg isRoot(q)$ then
24:     $updateBit(parent(q))$;
```

Details can be seen in the analysis of Subsection 4.3.3.

Procedure $updateBit(q)$ is called due to the push-into of any new element to the leaf stacks. Since the algorithm makes sure that when an element is pushed into the stack, its ancestors which match the query path from root to its corresponding query node are already in the stacks, the pushing of the element into leaf stack means that there must be a match to a path pattern query. Then the matching information for the correlative elements in the ancestor stacks need to be updated. Actually, it is propagated to the correlative elements from the stacks of leaf nodes to that of the root. However, for the inner query node, only if it has the exact descendant extension (This is achieved by checking the $bitVector$ of its own on Line 11.), its matching bit of its ancestors in the

```
Function $moveStreamToStack(q, S_q, ES_q)$
1: if $PC(parent(q), q)$ then
2:     if $PC_d(elem(ES_{parent(q)}, top), Next(S_q))$ then
3:         $push(ES_q, Next(S_q), 0, NIL)$;
4:         $bit(ES_{parent(q)}, 0, top).set$;
5:         return $true$;
6:     else
7:         return $false$;
8: else
9:     $push(ES_q, Next(S_q), 0, NIL)$;
10:    $bit(ES_{parent(q)}, 0, top).set$;
11:    return $true$;
```

parent stack can be set as “1”. Each time, if $PC(parent(q), q)$, matching information of at most 1 element in parent stack need to be updated. If $AD(parent(q), q)$, the matching information of the elements which correlate with the leaf element newly pushed into the stack, but not with the leaf element proceeding it need to be updated. While the last line of Procedure $cleanStack()$ makes sure that when the possible $outputList$ is propagated, the matching information for the elements in distinguished path stacks are reset and prepared to record the future matching information of the path pattern.

Line 4, 5 (Function $moveStreamToStack()$) of TwigContainment makes sure that only e_q which satisfy the path pattern query from root to q can be pushed into the stack. Line 5 push the element from the stream to the stack. After a element is pushed into the extended stack, the stream can be advanced. In the procedure $moveStreamToStack()$, we push the next elements in S_q into ES_q, and set the value of $bitVector$ as all “0”. This is due to the fact that when e is iterated, its possible correlative elements which matches to the nodes under q have not been accessed yet. Note that the value of $bitVector$ and $outputList$ may be changed later on in Procedure $updateBit(q)$ and $cleanStack()$ due to the appearance of new matching elements.

4.3.3 Analysis of TwigContainment

In this section, the proof of the correctness and completeness of the algorithm TwigContainment is given.

Lemma 4.3.2. *Consider the following fragment in Procedure $cleanStack()$:*

```
for $\forall q_c \in children(q)$ do
    $cleanStack(q_c)$;
while $topR(ES_q) < nextL$ do
    $\cdots$;
    $pop(ES_q)$
```

If in Algorithm 3, $q_{act} = q$ and e_q is cursor element of S_q, before e_q is pushed into the extended stack ES_q, the following properties hold:

(1) All the elements in stacks (from bottom to the top) are guaranteed to lie on a root-to-leaf path in the XML database.

(2) All the elements popped out from the ancestors stacks of q cannot be in the same solution with e_q and the elements following e_q in the streams.

(3) All the elements popped out from the descendant stacks of q cannot be in the same solution with e_q and the elements following e_q in the streams.

Proof. •

(1) According to Algorithm 3, the elements in streams are processed according to preorder. Procedure $cleanStack()$ is called recursively in preorder of the query nodes. From above fragment, we know that all the elements remaining in the stacks are those which end after e_q starts. Since elements in XML documents are nested. The remaining elements must end after e_q ends. Since elements start after e_q starts

haven not be accessed yet. Thus, after calling $cleanStack(root(Q), nextL(S_{q_{act}}))$, all the elements e in the stacks satisfies: $e.LeftPos < e_q.LeftPos < e_q.RightPos < e.RightPos$, i.e. they are in the same path with e_q.

(2) Let q'' be any ancestor of q in Q and $e_{q''}$ is popped out before e_q is pushed into ES_q. Then $e_{q''}.RightPos < e_q.LeftPos$. Assuming that $e_{q''}$ is in the same solution with e_q or the element following e_q, then $e_{q''}.RightPos > e_q.RightPos > e_q.LeftPos$, which is contradictory to the assumption.

(3) Let q'' be any descendant of q in Q and $e_{q''}$ is popped out before e_q is pushed into ES_q. Then $e_{q''}.RightPos < e_q.LeftPos$. Assuming that $e_{q''}$ is in the same solution with e_q or the element following e_q, then $e_{q''}.RightPos > e_{q''}.LeftPos > e_q.LeftPos$, which is contradictory to the assumption.

□

Lemma 4.3.3. *Algorithm 3 makes sure that all and only the elements e_q in S_q that satisfy the predicates between $root(Q)$ and q are pushed into the stacks.*

Proof. Firstly, it is necessary to prove that all the elements that satisfy the path pattern are pushed into the stacks. This can be proved by induction on the level of q. For the elements in $S_{root(Q)}$, they are pushed into the stacks directly according to Line 5 of Algorithm 3. The property holds. Suppose that the property holds for any node of level i in query. Let q be on the $i + 1$th level and let q' be its parent node. Assume that e_q be an element from the stream of S_q, which satisfies the query predicate from root to q. There must be an element $e_{q'}$ from $S_{q'}$ which on the path from root element to e_q and match the predicates from $root(Q)$ to q'. Obviously $e_{q'}$ is processed before e_q. Since q' is of level i, $e_{q'}$ must be pushed into $ES_{q'}$ according to inductive hypothesis. Obviously, $e_{q'}.LeftPos < e_q.LeftPos < e_q.RightPos < e_{q'}.RightPos$; Any elements $e_{q''}$ accessed between $e_{q'}$ and e_q satisfies $e_{q''}.LeftPos < e_q.LeftPos$. Thus $e_{q'}$ cannot be popped out from $ES_{q'}$ before e_q is pushed into stack ES_q. According to Line 4, e_q can be pushed into ES_q.

Next, we need to prove that if e_q does not satisfy the predicates from $root(Q)$ to q, it cannot be pushed into the stack. According to Algorithm 3, e_q can be pushed in to ES_q if and only if $ES_{parent(q)}$ is not empty after calling $cleanStack()$, and e_q, together with $top(ES_{parent(q)})$, satisfies the predicate between q and $parent(q)$. Obviously, for any element e_i remains in stack ES_i after this function call, its associated element in the $ES_{parent(i)}$ should remain in the stack as well. Thus, if $ES_{parent(q)}$ is not empty, the element in stacks must comprise the path pattern match for $top(ES_{parent(q)})$. And if e_q and $top(ES_{parent(q)})$ satisfy the predicate between q and $parent(q)$, there must be a pattern match for e_q in the stacks. This is contradictory to the assumption. □

According to Lemma 4.3.3, if an element is pushed into the extended stack of one of the leaf nodes, the element can match the path pattern query from the root to the leaf node.

Lemma 4.3.4. *Procedure $updateBit()$ makes sure that:*

(1) *If all matching bits of $bitVec(ES_q, Itm)$ are "1" in the stack of an inner node q, then the element $elem(ES_q, Itm)$ is in a match to the sub-query rooted at q;*

(2) *Let e_q be an element in the stack ES_q. If it has a match to the sub-query rooted at q, the corresponding $bitVector$ will be set as all "1" before it is popped out.*

Proof. •

(1) (Induction on the height of q.) The height of a leaf node is defined as 0; And the height of a internal node is defined as the largest height of its children plus 1. For the elements pushed into the leaf stack, although the associating $bitVector$ are not updated, they surely match the sub-query rooted at the leaf node. Suppose that

for the query nodes of height i, the property is verified. For the node q of height $i+1$, assume that e_q is in ES_q and has all "1" $bitVector$ matching information. From $updateBit()$ we know that $\forall q_c \in chilren(q)$, there must exists a element e_{q_c} which has all "1" $bitVector$ in ES_{q_c} and satisfies the predicate between q and q_c. q_c's height is at most i, thus, e_{q_c} is in a match to the sub-query rooted at q_c. And according to $updateBit()$, e_q and e_{q_c} ($\forall q_c \in children(q)$) satisfy the predicate between q and q_c. Thus e_q is in a match to the sub-query rooted at q.

(2) According to the first property of Lemma 4.3.2, e_q satisfies predicates from $root(Q)$ to q. Assume q' is a descendant node of q and $e_{q'} = corr(e_q, q')$. Thus, $e_{q'}$ must satisfy the predicates from $root(Q)$ to q' and be pushed into the stack. We prove by induction on the height of the query node. Assume q is of height 1. Once $e_{q'}$ is pushed into the stack, the $updateBit()$ is called and the corresponding bit of e_q will be set. Since all its descendant matches starts before it ends, the matching bits of e_q's $bitVector$ are set to be all "1" before it is popped. Assume the nodes of height i verify this property. Now let q is of height $i+1$. Then all the $bitVector$ of the child node matches of e_q should be set all "1" before popped out according to hypothesis. According to $updateBit()$, once they are set as all "1", through $updateBit(parent(q))$, the bit of q is set. Thus the property is verified on q.

□

According to the second property of Lemma 4.3.2, if a element e_q popped out with its corresponding $bitVector$ not being all "1", then it cannot be in a match to the sub-query rooted at q. We know that the elements in $S_{root(Q)}$ are pushed into $ES_{root(Q)}$; Then according to the two properties of Lemma 4.3.4, for any element $e_{root(Q)}$, if and only if it is in a match of Q, its corresponding matching bits in $bitVector$ can be set to be all "1" before it is popped out from $ES_{root(Q)}$. However, one element in the $ES_{root(Q)}$ may have multiple solution matches.

Lemma 4.3.5. *The procedure $cleanStack()$ can make sure that*

(1) *each solution e_{q_d} to the distinguished node q_d can be merged into the $outputList$ of the correlative element $e_{parent(q_d)}$ (if any) in the $ES_{parent(q_d)}$ before $e_{parent(q_d)}$ is popped out;*

(2) *the $outputList$ containing each solution e_{q_d} will not be dropped during query evaluation;*

(3) *and the elements e_{q_d} in the stream S_{q_d} which is not the solution cannot be output.*

Proof. •

(1) Let e_{q_d} be a result. Then e_{q_d} must match the predicates between $root(Q)$ and q_d (i.e. e_{q_d} has ancestor extension). According to Lemma 4.3.2, it must be pushed into the stack ES_{q_d} when it is iterated by the cursor. At the same time, e_{q_d} is in a partial solution of the sub-query rooted at q_d (i.e., it has descendant extension). According to Lemma 4.3.4, its matching bits in $bitVector$ must be set as all "1" after all its descendant extensions are iterated. Since procedure $cleanStack()$ are called in recursive order, e_{q_d} can be propagated to the $outputList$ of the correlative element in $ES_{parent(q_d)}$ before it is popped out.

(2) For this item, we need to prove that each solution e_{q_d} will be successfully propagated from the stack of the ith ancestor, denoted as ES_{q_i}, to that of the $i + 1$th ancestor, $ES_{q_{i+1}}$ (except $root(Q)$) during the evaluation and can be output successfully from $ES_{root(Q)}$. According to Procedure $cleanStack()$ and Procedure $nextMatch()$, e_{q_d} can be propagated if and only if there is a correlative element e_{q_i} which has descendant extension in ES_{q_i}. And e_{q_d} is propagated in the $outputList$ of the correlative element of e_{q_i}. We have two cases:

(a) There is only PC relationship between q_i and q_d; Thus, the partial solution between them associating e_{q_d} is one and only, i.e. the match traced by the parent pointer in each stack. Thus, e_{q_d} can be successfully propagated from ES_{q_i} to $ES_{q_{i+1}}$

(b) There exists AD relationship between q_i and q_d; If e_{q_d} is dropped, the reason is that, at certain ancestor level, j ($0 < j < i$), e_{q_d} is propagated in the $outputList$ of e_{q_j} (the lowest ancestor with descendant extension) whose correlative element e_{q_i} in ES_{q_i} has no descendant extension.In procedure $nextMatch()$, element e_{q_d} is merged into the $outputList$ of e'_{q_j} which also has a match to the whole path pattern containing q_d. If AD constrain is under q_i in the path leading to q_d, obviously, e'_{q_j} is also correlative to e_{q_d}. Otherwise, the path pattern between q_i and q_d is shown in Figure 4.7.(a) where q_k is the first query node followed by the AD constrain under q_i (k may equal to j). The dashed line and dotted line represent the part whose constrain can be ignored. The correlative element of e_{q_i} (e_{q_k}) is different to that of e'_{q_j} (e'_{q_k}) in ES_{q_k}. The match is shown in Figure 4.7.b. Since q_k followed by AD constrain in the path, e'_{q_k} is also correlative to e_{q_d}. So does e'_{q_i}. Because e_{q_d} is in a whole match, there must be a element which is correlative to e_{q_d} and has descendant extension in ES_{q_i}. So, e_{q_d} can be successfully propagated from ES_{q_i} to $ES_{q_{i+1}}$

Inductively, all the solutions can be output successfully.

(3) According to Lemma 4.3.3, in S_{q_d}, e_{q_d} which has no ancestor extension cannot be pushed into the stack. According to the first item, e_{q_d} which has no descendant extension cannot be propagated into the parent stack of q_d. And according to the second item, e_{q_d} which does not satisfy other pattern constrains cannot be

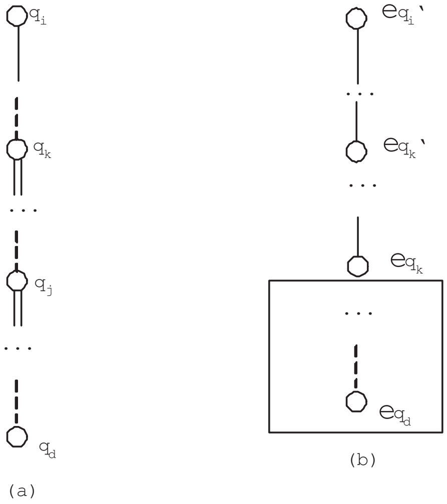


Figure 4.7: Path Pattern Match

propagated from the ith ancestor stack to the $i+1$th ancestor stack.

□

The Procedure $cleanStack()$ outputs the $outputList$ of the elements in the stack $ES_{root(Q)}$ before they are popped out if their $bitVector$ are all “1”. From Lemma 4.3.5, we can be sure that all the different matches to the distinguished node q_d are correctly output. So we have the following Theorem:

Theorem 4.3.6. *Given a twig query Q and an XML database D, Algorithm TwigContainment correctly returns all the answers for Q on D.*

It is noted that the distinguished nodes are propagated only to the lowest correlative elements in the parent stack. By doing this, the memory space and the answers which appear in multiple solution matches will not be output redundantly (one of the main problem of TwigStack). The correctness and the completeness of the algorithm is proved.

If the final answers are required be presented in sorted document order, in Procedure $CleanStack()$, when any element is popped from the stack of the root, we cannot directly output all elements in its $outputList$ (Line 25). Instead, its $outputList$ need to be merged into that of the next element in root stack. In general, the output of elements is blocked until all answers prior to them in the sort order can be computed.

When there are multiple distinguished nodes in the queries, algorithm TwigContainment should create the corresponding $outputList$ for each of them. We know that each $outputList$ associates with a element, then the merge join part are processed when the element is popped out and the $outputList$ is propergated to the element in the parent stack. However, a matches to one distinguished node can be joined with matches of other distinguished node at different level. For example, in Figure 4.6, if the query is $A[//C*]/B*$, then both (C_1, B_1) and (C_1, B_2) are solutions. Then for each branching node q which has more than one outgoing distinguished paths, when the $outputList$ of e_q is propogated, it should be merged to that of the parent stack element's as well as to that of the element's which is under e_q in ES_q if the corresponding constrain is AD relationship. It is important to note the differences between TwigStack and TwigCongtainment. TwigStack may output many path solutions that do not contribute to any final answers. However, TwigContainment guarantees that each output is one of the final answers.

Example 2. *We use the XML document and query in Figure 4.6 again to illustrate how to use $bitVector$ to avoid outputting "useless" elements. Table 1. traces the entire matching process by showing the $bitVector$ updates and the corresponding stack operations. Note that for this example, the previous algorithms (e.g. TwigStack and TwigStackList) will output a useless path solution (A_1, B_1), but TwigContainment only output one useful solution B_2.* □

Step	$cleanStack()$	$moveStreamToStack()$	$updateBit()$
1		$push(ES_A, A_1, 0, NIL)$	
2		$push(ES_B, B_1, 0, NIL)$	$(A_1, “10'', NIL)$
3	$(A_1, “10'', B_1)$ $pop(ES_B)$	$push(ES_A, A_2, 0, NIL)$	
4		$push(ES_B, B_2, 0, NIL)$	$(A_2, “10'', NIL)$
5	$(A_2, “10'', B_2)$ $pop(ES_B)$	$push(ES_C, C_1, 0, NIL)$	$(A_2, “11'', B_2)$
6	$pop(ES_C)$ output B_2 $pop(ES_A)$ $pop(ES_A)$		

Table 4.1: Matching Process for Example 2

4.4 TwigPrefix

In this section, the second novel algorithm, TwigPrefix is presented, which is inspired by the *extended Dewey* encoding method proposed in [74]. *Extended Dewey* is a prefix numbering scheme and encodes the element name under a specific parent context by using modulo function. A finite state transducer (FST) can be defined according to the XML schema to decode the encoding numbers along the path from the root to an element. Thus, from the *extended Deway* numbering of an element alone, the names of the all the elements in the path from the root to this element can be derived. The details of this element decoding method and the FST is introduced in Section 2.4.3. In the following section, the additional data structures and notations used in TwigPrefix is introduced first.

4.4.1 Data Structure

For each leaf node q_l in the twig query, there is a associating stream $\widetilde{S}_{q_l}$. The stream contains *extended Dewey* numbers of elements that match the node type q_l. The element numbers in the stream are sorted in the *ascending* lexicographical order (which is actually consistent with the pre-order traverse of the elements). The function $next(\widetilde{S}_{q_l})$ returns the *extended Dewey* number of the cursor element in the stream $\widetilde{S}_{q_l}$. Operation $advance(\widetilde{S}_{q_l})$ skips the pointers to the next elements. For two elements e_p and e_q [2] (p is an ancestor of q), Procedure $Rel_d(e_p, e_q)$ verifies whether they matches the path pattern between p and q.

Similarly, a twig query on XML can be represented with a small tree structure Q. Given a query node q, functions $LBA(q)$ and $HBD(q)$ return the *lowest branching ancestor* node of q and the *highest branching descendant* of q respectively if they exists (if q is a branching node, q itself is returned). For example, in Figure 4.6, $LBA(C) = A$. In addition, the self-explaining functions $isBranch(q)$ and $isTopBranch(q)$ is used to determine whether q is a branching node and the highest branching node accordingly. If q is a branching node, Function $dbl(q)$ returns the set of all branching nodes q_b and leaf nodes q_l under q s.t. there is no branching node between q and q_b, and between q and q_l. Function $isDist(q)$ and $isAnceDist(q)$ check whether q is the distinguished node or is an ancestor of the distinguished node.

TwigPrefix keeps a extended stack structure $\widetilde{ES}_{q_b}$ for each branching node q_b during execution too. Each item in stacks consists of a 4-tuple $(num(e_q), bitVector, outputList, ptrP)$, which has the similar property as that in stacks of TwigContainment. However, the $num()$ is the *extended Dewey* number, the size of $bitVector$ is $|dbl(q)|$ now. (With the dewey encoding method, the flag bit is not necessary.) And it should be noted that the $ptrP$ of each item in $\widetilde{ES}_q$ is pointed to its lowest correlative element in the stack of the $LBA(q)$ (if existing). Functions $elem(\widetilde{ES}_q, Itm)$, $bitVec(\widetilde{ES}_q, Itm)$, $outputL(\widetilde{ES}_q, Itm)$, $ptr(\widetilde{ES}_q, Itm)$ and $bit(\widetilde{ES}_q, q_i, Itm)$ ($q_i \in dbl(q)$) are defined similarly as those of TwigContainment. The maximal number of elements in each stack is no more than the max depth of the document. Furthermore, since only *branching* nodes have extended

[2] Here, e_p and e_q represent both the elements and the numbers of them

Algorithm 4 TwigPrefix

```
Input: $Q$ is a twig pattern query with distinguished node $q_d$
for $\forall q \in Q$ do
    $isLeaf(q) \Rightarrow locateMatchedElem(q)$;
while $\neg(end(root(Q)))$ do
    $q_{act} = getMinSource(root(Q))$;
    $cleanStack(root(Q), next(\widetilde{S}_{q_{act}}))$ ;
    $moveStramToStack(q_{act})$;
    $advance(\widetilde{S}_{q_{act}})$;
    $updateBit(q_{act})$;
    $locateMatchedElem(q_{act})$;
Procedure: $locateMatchedElem(q)$
{ Assume that the prefix of element $next(\widetilde{S}_q)$ is $n_1/n_2/\cdots/n_k$ }
  while $\neg((n_1/n_2/\cdots/n_k$ match path pattern query of$q)$ and $(n_k$ matches $q))$ do
      $advance(\widetilde{S}_q)$;
```

stack structures in TwigPrefix, a *responsible node* associated with the the distinguished node q_d, denoted as $resp(q_d)$ is defined as follows.

Definition 4.4.1 (Responsible Node). For a *distinguished* node q_d in query Q, its *responsible node* is defined as:

$$resp(q_d) = \begin{cases} HBD(q_d) & \text{if } HBD(q_d) \text{ exists ;} \\ LBA(q_d) & \text{otherwise, if } LBA(q_d) \text{ exists .} \end{cases}$$

4.4.2 Algorithm

The main algorithm of TwigPrefix is shown in Algorithm 4 and all stack operations are shown in Function $cleanStack(\, , \,)$ and $updateBit()$. The main idea of TwigPrefix is also to use $bitVector$ to precisely record the matching results and use $outputList$ to contain possibly matching elements. The procedures $locateMatchElem()$, is similar to that in algorithm TJFast[74].

```
Procedure $cleanStack(q, e)$
for $\forall q_c \in dbl(q)$ do
    if $isBranch(q_c)$ then
        $cleanStack(q_c, e)$;
while $elem(\widetilde{ES}_q, top)$ is not an ancestor of $e$ do
    if $isTopBranch(q)$ then
        if matching bits of $bitVec(\widetilde{ES}_q, top)$ are all "1" then
            output $outputL(\widetilde{ES}_q, top)$;
        else
            $Itm = nextMatch(q)$;
            Append $outputL(\widetilde{ES}_q, Itm)$ with $outputL(\widetilde{ES}_q, top)$;
    else if $q = resp(Q)$ and matching bits of $bitVec(q, top)$ are all "1" then
        if $q = HBD(q_d)$ then
            Append $outputL(\widetilde{ES}_{LBA(q)}, ptr(\widetilde{ES}_q))$ with $e_{q_d}$ correlated with
            $elem(\widetilde{ES}_q, top)$ and $elem(\widetilde{ES}_{LBA(q)}, ptr(\widetilde{ES}_q))$;
        else
            Append $outputL(\widetilde{ES}_{LBA(q)}, ptr(\widetilde{ES}_q))$ with $e_{q_d}$ correlated with
            $elem(\widetilde{ES}_q, top)$ and $e$;
        break;
    else if $isAnceDist(q)$ then
        if matching bits of $bitVec(q, top)$ are all "1" then
            Append $outputL(\widetilde{ES}_{LBA(q)}, ptr(\widetilde{ES}_q, top))$ with $outputL(\widetilde{ES}_q, top)$ ;
            break;
        else
            $Itm = nextMatch(q)$
            Append $outputL(\widetilde{ES}_q, Itm)$ with $outputL(\widetilde{ES}_q, top)$;
    $pop(\widetilde{ES}_q)$;
if $isTopBranch(q)$ then
    clear "1 " bit for items in stacks of distinguished path;
```

```
Procedure $nextMatch(q)$
for $Itm_1$ from top to bottom do
    if $elem(ES_q, q_c, Itm_1)$ satisfy pattern between $q$ and $q_c$, with $q_c \in dbl(q)$ in distinguished path then
        break;
return $Itm_1$;
```

The Procedure $end()$ of Algorithm 4 is the same as that in Algorithm 3.

Procedure $updateBit(q)$

```
if $isLeaf(q)$ then
    $bit(\widetilde{ES}_{LBA(q)}, q, Itm_2).set$;
else
    for $\forall Itm_1 \in \widetilde{ES}_q$ do
        if $bitVec(\widetilde{ES}_q, Itm_1)$ are all “1” then
            for $\forall Itm_2 \in \widetilde{ES}_{LBA(q)}$ do
                if $Rel_d(elem(\widetilde{ES}_{LBA(q)}, Itm_2), elem(\widetilde{ES}_q, Itm_1))$ then
                    $bit(\widetilde{ES}_{LBA(q)}, q, Itm_2).set$;
if $\neg isTopBranch(q)$ then
    $updateBit(LBA(q))$;
```

Procedure $moveStreamToStack(q)$

```
for $q_i$ in path from $root(Q)$ to $q$ do
    if $isBranch(q_i)$ then
        for all element $e$ matching $q_i$ in the $prefix(next(\widetilde{S}_q))$ do
            if $e$ is descendant of $elem(\widetilde{ES}_{q_i}, top)$ then
                $push(\widetilde{ES}_{q_i}, e, 0, NIL)$;
```

4.4.3 Analysis of TwigPrefix

Lemma 4.4.2. *Procedure $cleanStack()$ makes sure that When an element e_q is pushed into the extended stack $\widetilde{ES}_q$, the following properties hold:*

(1) All the elements in one stack (from bottom to the top) are guaranteed to lie on a root-to-leaf path in the XML database.

(2) All and only the elements e_q in $\widetilde{S}_q$ that satisfy the predicates between $root(Q)$ and q are pushed into the stacks.

(3) All the elements popped out from the ancestor branching node stacks cannot be in the same solution with e_q and the elements following e_q in the streams.

(4) All the elements popped out from the descendant stacks cannot be in the same solution with e_q and the elements following e_q in the streams.

Lemma 4.4.3. *Procedure $updateBit()$ makes sure that:*

(1) If matching bits of $bitVec(\widetilde{ES}_q, Itm)$ are all "1" in the stack of a inner node q, then the element $elem(\widetilde{ES}_q, Itm)$ is in a match to the sub-query rooted at q;

(2) Let e_q be an element in the stack $\widetilde{ES}_q$. If it has a match to the sub-query rooted at q, the corresponding $bitVector$ will be set as all "1" before it is popped out.

Theorem 4.4.4. *Given a twig query Q and an XML database D, Algorithm TwigPrefix correctly returns all the answers for Q on D.*

The proof of Lemma 4.4.2 and Lemma 4.4.3 and Theorem 4.4.4 are similar to that of TwigContainment. For the queries with more than 1 distinguished node, the output methods are similar to that of TwigContainment as well.

4.5 Time and Space Analysis

While the correctness and completeness hold for any given query, the I/O optimality holds only for the case where all *distinguished* nodes are optimal in Definition 4.2.8. Intuitively, this can be explained that when all *distinguished* nodes are optimal nodes, there are only *unbounded matching blocks* (see Theorem 4.2.9). Thus, TwigContainment and TwigPrefix are able to cache limited number of elements in $outputLists$ in the main memory and guarantee that each output elements in the two Procedures $cleanStack()$ for TwigContainment and TwigPrefix respectively belong to the final query solutions.

Theorem 4.5.1. *Consider an XML database D and a twig query Q where all distinguished nodes are optimal nodes. The worst case I/O complexity of TwigContainment and TwigPrefix is linear in the sum of the sizes of input and final query solution lists. The worst-case space complexity is linear in the maximal depth in D.*

Proof. According to the theoretical analysis of the algorithm, only the matches of the distinguished nodes which contribute to the final answers are output by Algorithm TwigContainment and TwigPrefix. Thus, the worst case I/O complexity is linear in the sum of the sizes of input and final query solution lists.

The key factor of the proof of the space complexity is to show that when all distinguished nodes are optimal nodes, given any stacks ES_q (or $\widetilde{ES_q}$), the number of the elements in its $outputList$s are no more than the max depth of the XML document. It is shown that Algorithm TwigContainment and TwigPrefix only store the matches to the distinguished node in the $outputList$. According to Theorem 4.2.9 and Theorem 4.2.10, there is no UMB on the stream of optimal distinguished nodes. And according to Lemma 4.3.5, one match of the distinguished node appears in at most one $outputList$ in any stack. The stack size is no longer than the maximum length of the XML documents. Thus the lemma is proved. □

When the main memory is extremely small and the query document is extremely large, if the distinguished node is not optimal, both TwigPrefix and TwigContainment cannot guarantee that all the elements in $outputList$ can be fit in the main memory. In this case, some elements in $outputList$ should be output as intermediate results. However, this is a rare practical occasion. In the next section, it is shown that for a large query class, even in the constraints of limited memory, TwigPrefix and TwigContainment guarantees that each output intermediate element belongs to final solutions. In sum, as the evidence of the generality of the theory on matching block, two algorithms TwigPrefix and TwigContainment are proposed which are based on different numbering scheme, but identify the same optimal query class to fulfill the results of Theorem 4.2.9 and 4.2.10. However, as shown in the next experimental part, although TwigPrefix and TwigContainment share the same query class for optimality, for the case of a non-optimal query, two algorithms may output different number of intermediate results due to the discrepancy of their numbering schemes.

4.6 Performance Study

In this section, extensive experimental study of TwigContainment and TwigPrefix is performed on real-life and synthetic data sets. The results verify the effectiveness, in terms of accuracy and optimality, of the TwigContainment and TwigPrefix as holistic twig join algorithms for large XML data sets. These benefits become apparent in a comparison to previously proposed three algorithms TwigStack [20], TwigStackList [73] and TJFast [74]. Overall, this empirical study indicates that TwigContainment and TwigPrefix fully exploit the key observation for *distinguished* nodes and thus significantly outperforms the existing holistic join algorithms. In addition, it also shows that TwigPrefix outperforms TwigCongtainment with respect to I/O cost and main memory requirement.

4.6.1 Experiment Settings and Datasets

Algorithms TwigStack, TwigContainment and TwigPrefix are implemented in JDK 1.4 using the file system as a simple storage engine. The codes of TwigStackList and TJFast come from authors of original papers [73, 74]. The reason that these three algorithms chosen for comparisons is that TwigStack, TwigStackList and TJFast are optimal for different query class. TwigStack is a well-known holistic twig algorithm, which is very efficient when query contains only AD relationships. TwigStackList extend TwigStack by adding list structure and thus identify a larger optimal query class. Finally, TJFast is based on a variant of prefix numbering scheme. It is claimed to significantly reduce I/O cost by accessing only numbers of *leaf* query nodes.

The experiments are all conducted on a workstation with Intel Pentium IV 1.7GHz

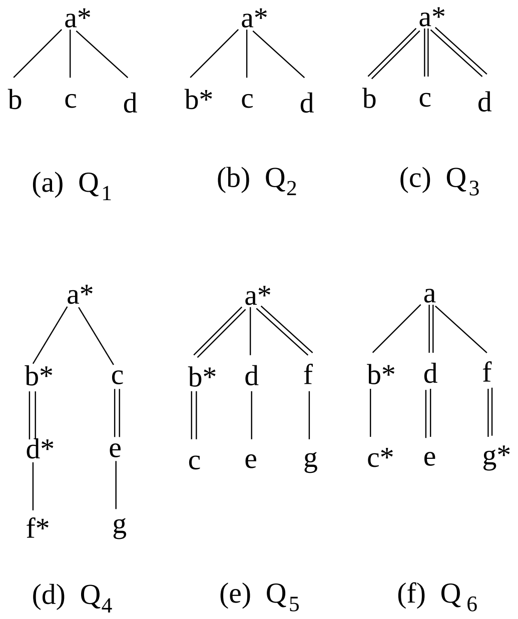
CPU and 512M of RAM. The operating system is windows XP. To offer a comprehensive evaluation of the new algorithms on different query types and on data with different features, both synthetic dataset and real XML data are used. The synthetic dataset is generated randomly. There are totally 7 labels A_1, A_2,..., A_7 in the dataset and labels are assigned uniformly from them. Two real datasets $DBLP$ and $TreeBank$ [16, 109] are used since they have different characteristics. $DBLP$ is a broad and shallow document, but $TreeBank$ has very deep recursive structure. Table 4.2 summarizes the dataset characteristics.

	Synthetic	$DBLP$	$TreeBank$
Size(MB)	8.8	130	82
Elements(million)	1.0	3.3	2.4
Max/Avg Depth	12/6.1	6/2.9	36/7.8

Table 4.2: Character of the Test Data Sets

In order to compare the performance of different algorithms under different workloads, a set of queries is designed, which have different features in terms of twig structures and *distinguished* nodes. All queries tested for random data sets are shown in Figure 4.8. In particular, Q_1, Q_2 contain only PC relationships, while Q_3 contains only AD relationships and Q_4,Q_5,Q_6 have different combinations of both PC and AD relationships. All queries proposed to $TreeBank$ and $DBLP$ data are shown in Table 4.3. In particular, Q_7, Q_8 and Q_{10} have single *distinguished* nodes and other queries have multiple *distinguished* nodes. Note that in Q_{12}, all nodes are *distinguished* ones. We use this query to show that even in the case when all queries nodes are distinguished nodes, our algorithms still outperform previous methods.

The performance measurement includes number of intermediate results, memory size and processing time in the experiments. The number of intermediate elements evaluates the total number of intermediate elements, which reflects I/O costs. The measurement of varying main memory size is used to test whether algorithms perform well in the case of



(a) Q_1 (b) Q_2 (c) Q_3

(d) Q_4 (e) Q_5 (f) Q_6

Figure 4.8: Queries for Synthetic Data

	Dataset	Query
Q_7	DBLP	//article[.//cdrom]//author*
Q_8	DBLP	//inproceedings[author]//title/sup*
Q_9	DBLP	//inproceedings*[author]//title/sup*
Q_{10}	TreeBank	/S*[.//VP/IN]//NP
Q_{11}	TreeBank	PP[.//IN*]/NP/VBN*
Q_{12}	TreeBank	PP*[.//IN*]/NP*/VBN*
Q_{13}	TreeBank	VP*[NN]/S*
Q_{14}	TrreBank	S[ADJP*]/PP[.//NP]//IN

Table 4.3: Queries for DBLP and TreeBank Data

limited main memory. The total execution time is obtained by averaging the time elapsed to answer a query with six consecutive runs, discarding the best and worst performance results.

4.6.2 Algorithms Based on Containment Numbering

In this section, the performances of the algorithms TwigStack, TwigStackList and TJ-Fast on the real and synthetic data sets are presented. In the first set of experiments, the main memory size is limited to 10K, to compare the performance of algorithms under the constraints of a small main memory. Figure 4.9 shows the query performance in terms of response times (in seconds) and Figure 4.10 shows the number of output elements for different queries. TwigContainment is distinctly more efficient than the other algorithms for all six queries. This is due to the fact that TwigContainment output less "useless" intermediate results. With the limited memory setting, TwigStack and TwigStackList have to output most of intermediate path matches to the second memory and reload them in the second phase for merging. However TwigContainment selectively cache limited elements in the $outputList$ instead of outputting many useless intermediate elements. This result suggests that under the constraints of limited memory, TwigContainment can efficiently utilize the small main memory and achieve better performance than TwigStack and TwigStackList.

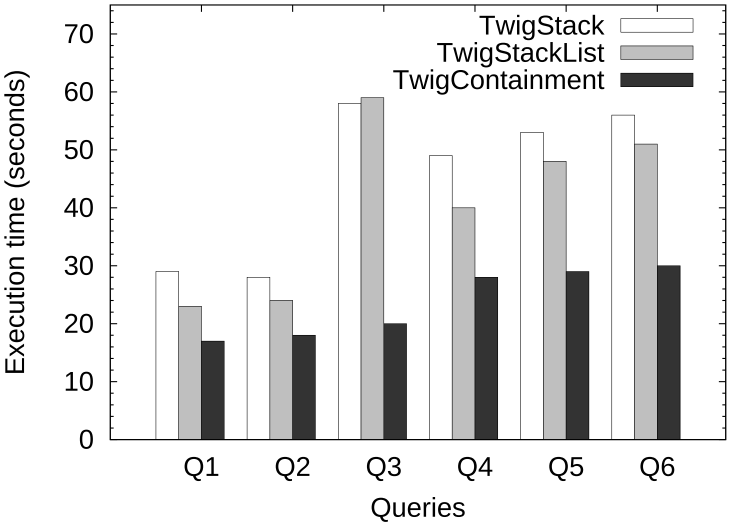


Figure 4.9: Execution Time (Synthetic)

Figure 4.11 and Figure 4.12 illustrate the performance of the algorithms on different size of main memory. Figure 4.11 shows the number of output elements of the three algorithms for query Q_1 where the number of elements allowed to be cached in main memory varied from 10K to 50K. Figure 4.12 is the result for Q_6. The two figures

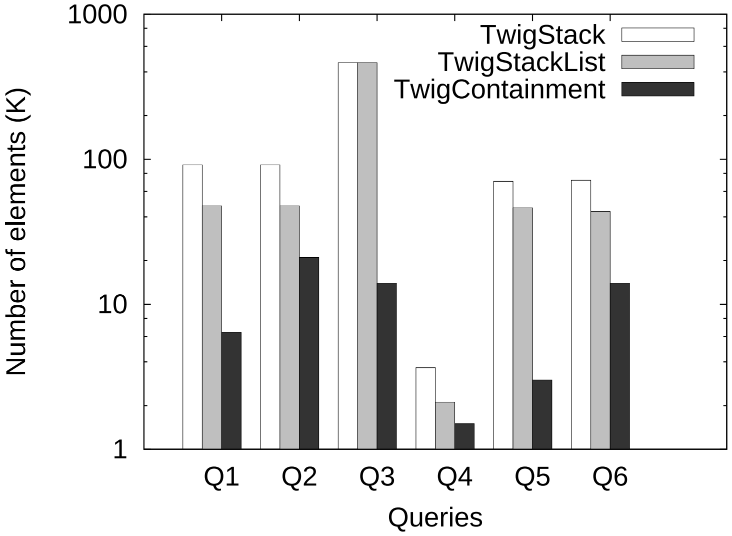


Figure 4.10: Output Element(Synthetic)

show that the output elements by TwigContainment is always much less than that of TwigStack and TwigStackList. In particular, for Q_1, with the increasing of the size of the available main memory, the output size of TwigStack and TwigStackList decreases linearly. The reason is that TwigStack and TwigStackList cache the intermediate results in the main memory and reduce the number of output elements. But the output of TwigContainment remains the same and equals to the final result size. This result confirms to Theorem 4.2.9, i.e., TwigContainment is an optimal algorithm for the queries where the root is the only distinguished node. For query Q_6, all algorithms are not optimal according to the theoretical analysis. But TwigContainment still output much less elements than TwigStack and TwigStackList. Finally, note that when the number of cached elements reaches 30K, TwigContainment does not output any useless elements for this data. It means that such main memory size is enough to hold the all the uncertain elements in the $outputLists$.

The next experiment is to compare the performance of three algorithms on $TreeBank$ and $DBLP$ datasets. Figure 4.14 and Figure 4.13 show the results of the time consumed and the number of output elements. For the execution time, Q_7, Q_8, Q_9, and Q_{10}

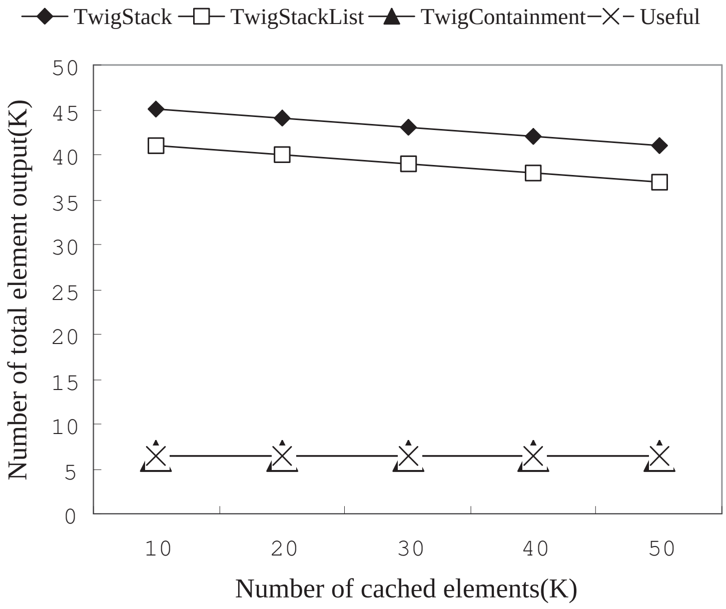


Figure 4.11: Output with varying memory (Q1)

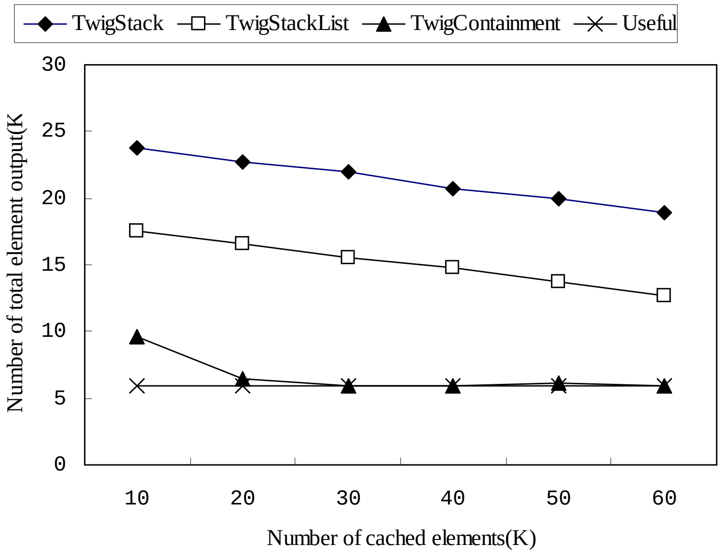


Figure 4.12: Output with varying memory (Q6)

TwigStackList use much less time than that of TwigStack and comparable to TwigContainment. For Q_{11}, Q_{12}, Q_{13} andQ_{14}, the consumed time of TwigStackList is significantly greater than that of TwigContainment. Again, the effect of the reduction of I/O cost in TwigContainment makes this algorithm superior to TwigStack and TwigStackList, reaching up to 51% improvement in execution time for all queries.

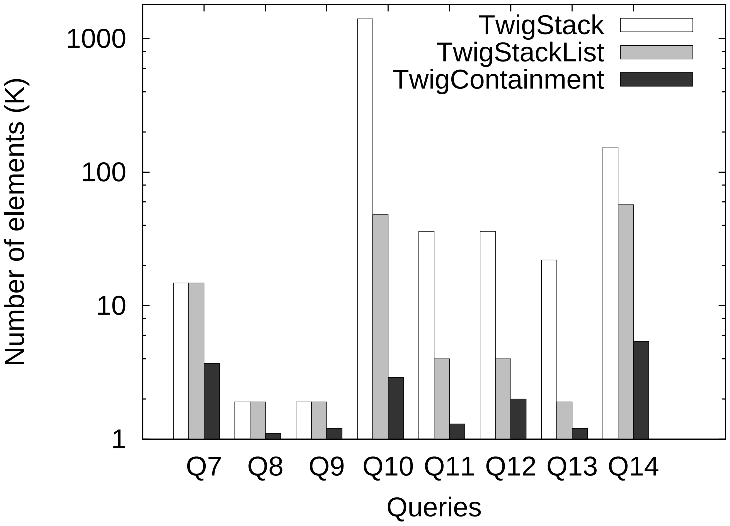


Figure 4.13: Output Element(real)

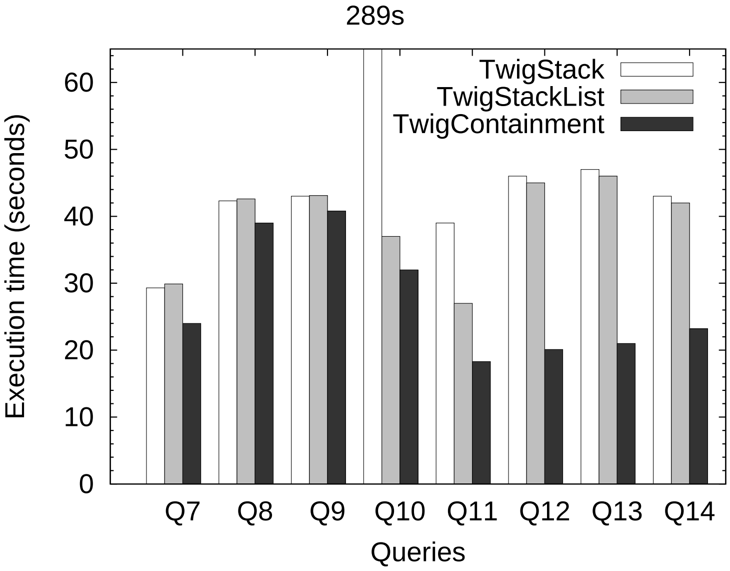


Figure 4.14: Execution Time (real)

Table 4.4 reports a comparison among the three algorithms about the number of output elements only for the *distinguished* nodes. The surprising result is that, for $DBLP$ data (Q_7-Q_9), three algorithms output the same number of elements for the *distinguished* node. This is due to the fact that $DBLP$ is a rather regular dataset without recursive structure. In contrast, for $TreeBank$ data, which is deeply recursive, TwigStack outputs large number of "useless" elements. For example, to query Q_{10}, TwigStack output 368983 elements, but only 10675 of them are in the final answers. Notice that for TwigContainment, the numbers in Table 4.4 is the same as that of the total output elements, but for Twigstack and TwigstackList, these numbers are much fewer than that of the total output elements.

	Q_7	Q_8	Q_9	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}
TwigStack	3722	605	1166	368983	13790	21298	23616	32928
TwigStackList	3722	605	1166	10675	1586	2882	470	5941
TwigContainment	3722	605	1166	10675	1317	2395	118	5446
Results	3722	605	1166	10675	1317	2395	118	5446

Table 4.4: Number of Output Elements for the Distinguished Node (Real)

4.6.3 Algorithms Based on *Extended Dewey* Numbering

In this section the performances of TJFast and TwigPrefix are compared. Both algorithms are based on the *extended Dewey* Numbering schemes. The queries shown in Figure 4.8 are also utilized over the synthetic datasets.

Figure 4.15 and Figure 4.16 show the number of elements output and the execution time. As shown from these results, TwigPrefix is more efficient than TJFast for all queries. These results reaffirm the effectiveness of the new algorithms.

Finally, TJFast and TwigPrefix are compared over $DBLP$ and $TreeBank$ dataset. The results are shown in Figure 4.17 and Figure 4.18. For all queries, TwigPrefix is again more efficient than TJFast.

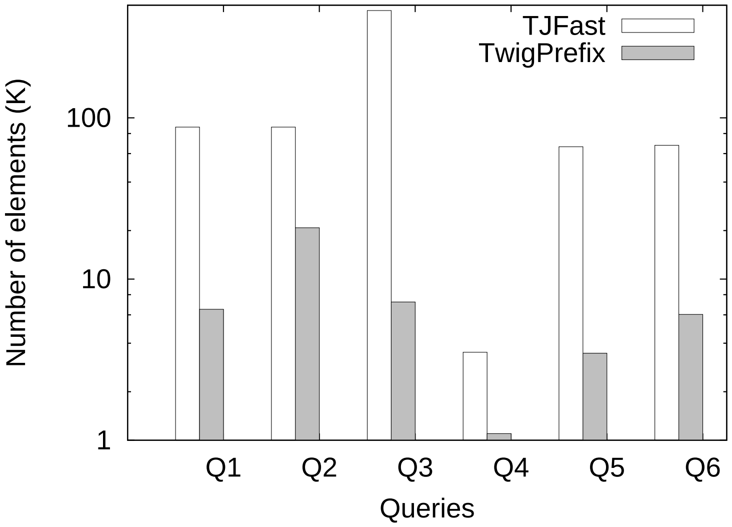


Figure 4.15: Output elements (Syn)

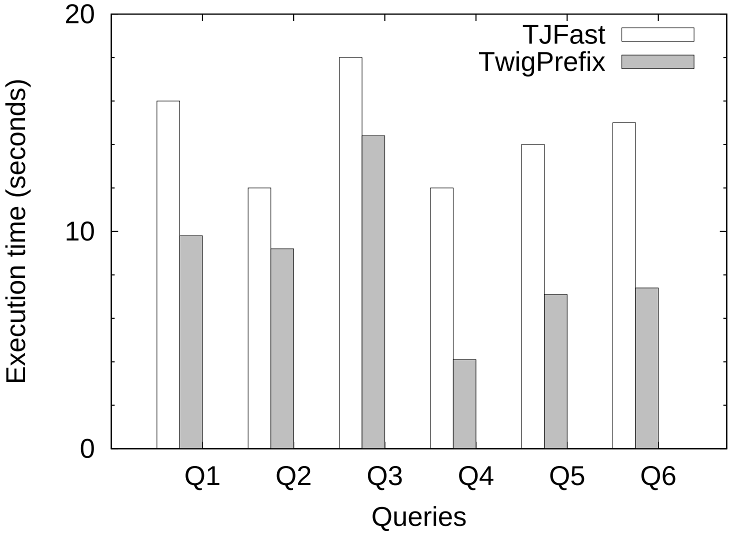


Figure 4.16: Execution Time (Syn)

4.6.4 Comparison between TwigContainment and TwigPrefix

In this section, we compare the performance between the two new algorithms with the memory size setting to be 10K. Figure 4.19 shows the CPU and I/O cost comparison. We can see that for synthetic data set, TwigPrefix outperforms TwigContainment for query Q_3 and Q_6 in terms of output element size. That is because the synthetic data re-

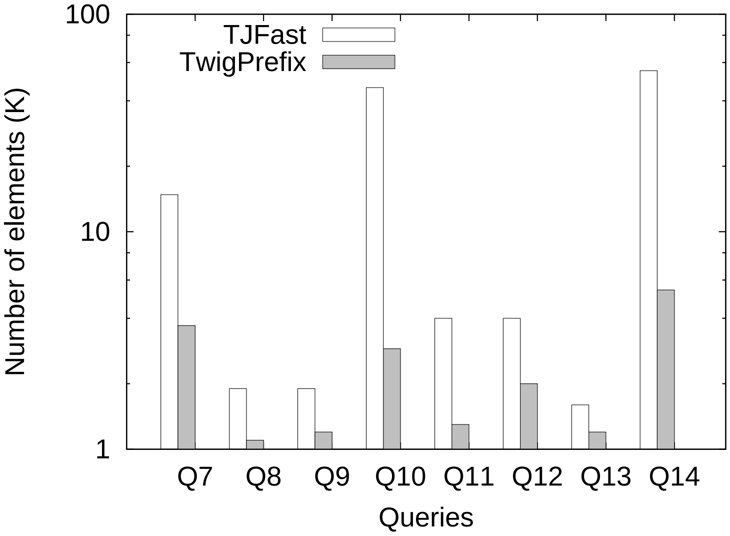


Figure 4.17: Output elements (real)

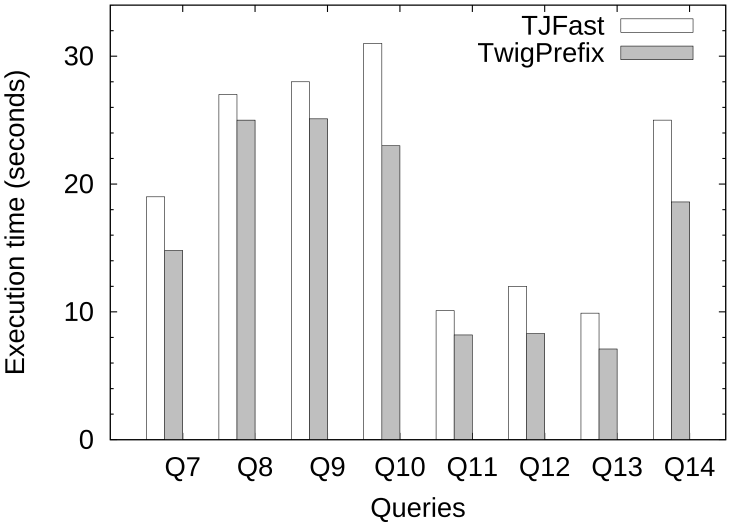


Figure 4.18: Execution Time (real)

curse deeply and the *Extended Dewey* numbering scheme can encode the element more succinctly and utilize the memory more efficiently. Thus more element can be cached in the memory. But for real dataset, both TwigContainment and TwigPrefix output the results only , and thus both are optimal. In the mean while, TwigPrefix obviously outperforms TwigContainment in terms of CPU cost. This is due to the fact that *Extended*

Dewey numbering can encode exactly the whole path for each element and accordingly, the query processing can be up to 7 times faster than TwigPrefix.

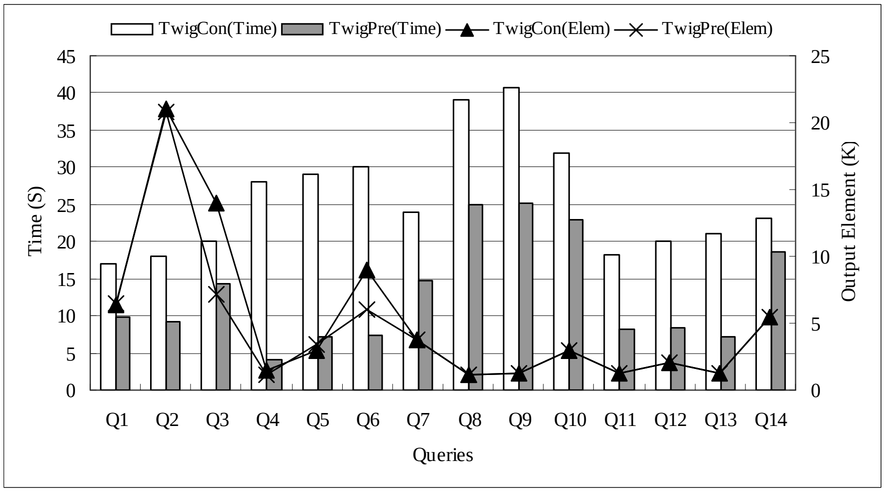


Figure 4.19: CPU and I/O Cost Comparison

As analyzed in Section 4.2, Section 4.4.3 and Section 4.5, the two new algorithms can guarantee the optimality for all kinds of queries if the available main memory is large enough. Table 4.5 shows that the max number of elements that should be stored in the main memory to guarantee the optimality of TwigContainment and TwigPrefix for synthetic data. There are two interesting findings:

(1) Comparatively speaking, TwigPrefix outperforms TwigContainment since it stores fewer elements in the main memory on all queries than TwigContainment does. This is due to the difference of numbering schemes in these two algorithms. *Extended Dewey* numbering scheme allows TwigPrefix to see the whole path by accessing only one element and therefore avoids storing redundant elements in the main memory.

(2) The number of elements that is needed to store in main memory for TwigPrefix is always small for all queries (e.g. the max number is 622, only about 6K Bytes memory). As mentioned in Section 4.4, we can deliberately design queries which

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6
TwigContainment	176	575	184	5453	253	26939
TwigPrefix	4	14	4	187	240	622

Table 4.5: Number of Required Cached Elements (Syn)

	Q_7	Q_8	Q_9	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}
TwigContament	4	7	8	13	54	82	10	17
TwigPrefix	3	4	5	3	19	82	4	7

Table 4.6: Number of Required Cached Elements (Real)

require TwigPrefix to cache a large number of elements such that they cannot be fitted into the small main memory. However, the empirical results show that even for the datasets which have the very deeply recursive structure, such as the synthetic dataset and $TreeBank$, it is not easy to find such unnatural queries to show the non-optimality of TwigPrefix.

Table 4.6 shows the max number of elements cached in the main memory for TwigPrefix and TwigContainment algorithms to guarantee the optimality. Interestingly, unlike the results in Table 4.5, the numbers of cached elements in both TwigPrefix and TwigContainment are very small. Therefore, for real datasets, even TwigContainment can guarantee that each output element belongs to final answers under the constraint of small available main memory.

From the above experimental results we can see that both TwigContainment and TwigPrefix have high performance on both the synthetic dataset and real dataset. And the main reason for the better performance of TwigPrefix is due to the encoding scheme. However, *Extended Dewey* encoding scheme is not always feasible in the practical application. For example, it cannot be applicable to the streaming dataset when the schema is not available. TwigContainment and reginal encoding is suitable for that situation.

4.7 Conclusion

XML data is rich in structure; and this calls for efficient structure join algorithms in order to facilitate XML query processing. In this chapter, the issue of XML twig pattern matching is studies. The critical observation is that, in most applications, only the result bindings of contain selected (*not all*) nodes are required. The theoretical analysis shows that the sub-optimality of previous holistic twig algorithm is due to the *bounded* or *unbounded matching block* (BMB and UMB). It is also analytically shown that the UMB that involves only undistinguished query nodes should not lead to the non-optimality of holistic twig algorithms.

Based on these analysis, two novel algorithms are proposed in this chapter. They are based on the *containment* and *prefix* numbering schemes respectively. These two algorithms not only avoid the output of elements for *undistinguished* query nodes, but also give the guarantee to the optimality for a much larger query class. The efficiency of these algorithms lies in the fact that the matching information, instead of different matches of the non-distinguished node is necessary. An excellent example is that two algorithms guarantee the optimality for any query with the *root* being the *distinguished* node, regardless of the combinations of PC and AD relationships within the query.

Chapter 5

Conclusion

In this chapter, we summarize the contributions of this thesis and discuss the future work on the similarity queries and pattern queries based on our methods.

5.1 Main Contribution

In this thesis, we extensively studied how to enhance two core operations on XML data, i.e., the similarity query and the pattern query on XML data. Similarity search is to find all objects in the database which are within a given distance from a given object (range query) or to the k most similar objects in the database which are closest in distance to a given object (k-NN query). While XML twig pattern query is to identify all the matches of the query nodes in data, which is actually a mapping from the query nodes to the elements of a certain XML data s. t. the predicates specified by the query nodes and the structural relationship depicted by the edges of the query nodes can be satisfied by their respective images under the mapping.

In this thesis, we propose a new distance between XML data. The measure function is based on the transformation of XML data into miniature structural feature vectors which combines the structural and content information conveyed by the node label. These miniature structure captures the local pattern of each data and the vector elements together describe the whole features of the XML tree structure. Each object is transformed to a sparse vector with $|T|$ non-zero items. The L_1 distances between the vectors are proved to be the lower bounds of the edit distance between the original tree structures. The intuition here is similar to that of q-gram methods solving approximate string matching problem. Thus, the original tree edit distance space is transferred to the vector space with L_1 norm distance.

We design and analyze the algorithms to embed the lower bounds into multi-step framework to solve the similarity search problems. The computation of the distance on the vector is only $O(|T|)$ for each comparison. With this lower bound, most of the computation of the real distance, with time complexity of

$$O(|T1||T2|min(depth(T1), leaves(T1))min(depth(T2), leaves(T2)))$$

, can be filtered. Like the q-gram methods which are used to processing similarity search on sequence data, our methods can be generalized according to different dataset characteristics. The comprehensive performance study experiments show that our methods are both I/O and CPU efficient.

For the XML twig query processing, firstly, we theoretically analyze the reason of the sub-optimality of previous algorithms and show that the existence of *matching blocks* on join data streams is the main cause. Previous algorithm suffers the existence of both the bounded and unbounded matching blocks. However, the research in this thesis demonstrates that unbounded matching block which involves the *existential* nodes should not result in the non-optimality of holistic algorithms. In addition, an unbounded matching block involving *distinguished* nodes also can be efficiently processed in most cases by selectively caching elements in the main memory.

Based on the theoretical analysis, we propose two novel algorithms TwigContainment and TwigPrefix based on *containment* and *prefix* numbering schemes respectively. The new algorithms employ the *bit vector* and *output list* structures to store information with bounded spaces to solve the unbounded matching blocks involving *distinguished* nodes. It is proved that the new algorithms identify a much larger I/O optimal query class. Because the theories are developed independent of any specific labeling scheme, these two algorithms have the same optimal query class. Finally, the new algorithms adopt a novel framework for holistic twig pattern matching. They makes one pass on the input data and directly output the matching elements of the distinguished node, without postprocessing phrase to do projection. The extensive experimental studies on synthetic and real datasets for performance comparison is presented in this thesis. The results show that TwigContainment and TwigPrefix outperform all tested previous methods. Moreover, although TwigContainment and TwigPrefix have the same optimal query class, the experimental results show that TwigPrefix outperforms TwigContainment in terms of the I/O cost and the total execution time.

5.2 Future Work

In this section, we propose several possible future work area based on the studies presented in this thesis.

5.2.1 Integrate XML documents

In order to integrate XML data from different sources, approximate matching method for trees is needed. For most of the data-centric XML document, the orders among siblings is not closely related to the information conveyed. Thus the approximate matching should be based on rooted, unordered, labeled trees. The distance function presented in this thesis is on ordered trees. However, the methods can be extended by using a canonical form representation for labeled rooted unordered trees [121]. From a rooted unordered tree we can derive many rooted ordered trees, we can uniquely select one as the canonical form to represent the corresponding rooted unordered tree. Thus, the (q-level) binary branch distance can be extended to measure unordered trees as well. Through the q-level) binary branch vector representation, the XML approximate join can then be transformed to equality join on vectors.

5.2.2 Incrementally Maintain Indexes for Similarity Search

The similarity query processing methods proposed in this thesis is not utilizing any indexing structure currently. However, indexes of the positional miniature structure features (q-level binary branches) can be constructed to prune the search spaces. Furthermore, XML documents may be updated constantly especially for the scientific data conveyed by XML [63, 82, 62]. The similarity search methods proposed in this paper is based on static XML data. It cannot be extended directly to process the dynamic dataset. However, building the incrementally maintained index is possible since each edit operation only have a local effect on the index. Thus, based on the index, the efficiency and effectiveness of similarity search processing can be improved further.

5.2.3 Future Work for Pattern Query on XML Data

The observation and theory made in this work shed new light on many related works. Recently, there appears some efforts to solve the queries with preceding, preceding-sibling, following, following-sibling axes [75], "NOT" predicate [120], "OR" predicate [53] and for XML documents based on graph data model (i.e. TwigStackD [26]. In this thesis, the most research work are focused on the XQuery expressions with child and descendant axes. In the future, the work can be extended to solve the other axes queries easily.

Yet, recently, some researchers proposed that the FOR, LET, WHERE and RETURN clause of XQuery are of different semantics, and it is better to matching these expressions as a whole in terms of the generalized tree pattern (GTP) [27].

$$\begin{array}{ll} \text{FOR} & \$b\,\text{IN}\,//A/B[//D] \\ \text{LET} & \$c\ :=\ \$b//C \\ \text{RETURN} & \$b,\ \$c \end{array} \tag{5.1}$$

For example, In the above XQuery, the node C in the above query is optional, since according to the semantics of XQuery statement, any expression in the LET or RETURN clauses is optional. That means element which matches node B can be a result even without any descendant C element. And the matches of C node must be grouped together under their common B ancestor match since in a LET clause, the variable only takes one value, a single item or a sequence. In the future work, solutions can be proposed to answer the challenges proposed by this generalized tree pattern query.

Furthermore, query processing methods based on indexed documents (XB-tree [20] and XR-tree index [55] indexes) can also be explored in the future work.

Bibliography

[1] Xml path language (xpath), `http://www.w3.org/TR/xpath`, Nov 1999.

[2] Xquery 1.0: An xml query language, `http://www.w3.org/TR/xquery/`, Jun 2006.

[3] S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *Proc. 19th Int'l Conf. Very Large Data Bases*, pages 73–84, 1993.

[4] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semistructured data. volume 1, 1996.

[5] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web from Relations to Semistructured Data and XML*. Morgan Kaufmann Publisher, 1999.

[6] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Representing and querying XML with incomplete information. *12th PODS conference 2001*, 2001.

[7] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In *VLDB*, Roma, Italy, September 2001.

[8] C. C. Aggarwal, J. L Wolf, and P. S. Yu. A new method for similarity indexing of market basket data. In *SIGMOD*, pages 407–418, 1999.

[9] Akutsu and Halldorsson. On the approximation of largest common subtrees and largest common point sets. *TCS: Theoretical Computer Science*, 233, 2000.

[10] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, page 141. IEEE Computer Society, 2002.

[11] Dongwon Lee Angela Bonifati. Technical survey of XML schema and query languages. In *(Submitted for journal publication)*, 2001.

[12] Nikolaus Augsten, Michael H. Böhlen, and Johann Gamper. Approximate matching of hierarchical data using pq-grams. In *VLDB*, pages 301–312, 2005.

[13] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[14] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: An index structure for high-dimensional data. In *VLDB*, 1996.

[15] Elisa Bertino and Won Kim. Indexing techniques for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):196–214, June 1989.

[16] Dblp bibliographies. `http://www.informatik.uni-trier.de/ley/db/`.

[17] Philip Bille. A survey on tree edit distance and related problems. *TCS: Theoretical Computer Science*, 337, 2005.

[18] Philip Bohannon, Juliana Freire, Prasan Roy, and Jerome Simeon. From XML schema to relations: A cost-based approach to XML storage. In *ICDE conference*, page 64, 2002.

[19] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language XML, `http://www.w3.org/TR/REC-sml`, Oct 2000.

[20] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD Conference*, pages 310–321. ACM, 2002.

[21] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *ACM-SIGMOD*, pages 505–516, 1996.

[22] Barbara Catania, Wen Qiang Wang, Beng Chin Ooi, and Xiaoling Wang. Lazy XML updates: Laziness as a virtue of update and structural join efficiency. In *SIGMOD Conference*, pages 515–526, 2005.

[23] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *WebDB*, 1999.

[24] Edgar Chavez and Gonzalo Navarro. Towards measuring the searching complexity of metric sapces. In *Proc. of the Mexican Computing Meeting*, pages 969–978, 2001.

[25] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2):26–37, June 1997.

[26] Li Chen, Amarnath Gupta, and M. Erdem Kurul. Stack-based algorithms for pattern matching on DAGs. In *VLDB*, pages 493–504.

[27] Songting Chen, Hua-Gang Li, Junichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selcuk Candan. Twig^{2}stack: Bottom-up processing of generalized-tree-pattern queries over XML documents. In *VLDB*. ACM, 2005.

[28] Ting Chen, Jiaheng Lu, and Tok Wang Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *SIGMOD Conference*, pages 455–466. ACM, 2005.

[29] Weimin Chen. New algorithm for ordered tree-to-tree correction problem. *Journal of Algorithms*, 40(2):135–158, 2001.

[30] Zhiyuan Chen, H. V. Jagadish, Flip Korn, Nick Koudas, S. Muthukrishnan, Raymond Ng, and Divesh Srivastava. Counting twig matches in a tree. In *Proc. of 17th Int'l Conf. on Data Engineering*, pages 595–604, 2001.

[31] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient structural joins on indexed XML documents. In *VLDB*, pages 263–274, 2002.

[32] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. APEX: an adaptive path index for XML data. In *ACM SIGMOD conf.*

[33] Gregory Cobena, Serge Abiteboul, and Amélie Marian. Detecting changes in XML documents. In *ICDE*, 2002.

[34] D. Comer. The ubiquitous B-tree. *ACM Computing Survey*, 11(2):121–137, 1979.

[35] M. Consens and T. Milo. Optimizing queries on files. In *SIGMOD Conference*, 1994.

[36] Brian F. Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In *Proceedings of the 27th In-*

ternational Conference on Very Large Data Bases(VLDB '01), pages 341–350, September 2001.

[37] S. J. DeRose. Xquery: A unified syntax for linking and querying general XML. In *WWW The Query Language Workshop (QL)*, 1998.

[38] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1999.

[39] Paul F. Dietz. Maintaining order in a linked list. In *STOC*, pages 122–127, 1982.

[40] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele, and Till Westmann. Natix: A technology overview. In *NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*.

[41] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDBMS. *Bulletin of the Technical Committee on Data Engineering*, pages 27–34, September 1999.

[42] XML for Publishers and Printers (XPP). `http://www.xyvision.com/xpp.asp`, 2002.

[43] Z. Galil and K. Park. An improved algorithm for approximate string-matching. *Automata, Languages and Programming (ICALP'89), Lecture Notes in Compute Science*, 372:394–404, 1989.

[44] Minos N. Garofalakis and Amit Kumar. XML stream processing using tree-edit distance embeddings. *ACM Trans. Database Syst*, 30(1):279–332, 2005.

[45] Roy Goldman, Jason McHugh, and Jennifer Widom. From semistructured data to XML: Migrating the lore data model and query language. In *WebDB (Informal Proceedings)*, pages 25–30, 1999.

[46] Roy Goldman and Jennifer Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. pages 436–445. VLDB, 1997.

[47] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 327–340, 2001.

[48] Sudipto Guha, H. V. Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. Approximate XML joins. In *SIGMOD Conference*, pages 287–298, 2002.

[49] Arvind Gupta and Naomi Nishimura. Finding largest subtrees and smallest supertrees. *Algorithmica*, 21(2):183–210, 1998.

[50] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[51] HR-XML. `http://www.hr-xml.org`, 2001.

[52] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V. S. Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. Timber: A native xml database. *VLDB Journal*, 11(4):274–291, 2002.

[53] Haifeng Jiang, Hongjun Lu, and Wei Wang. Efficient processing of XML twig queries with OR-predicates. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data 2004, Paris, France, June 13–18, 2004*, pages 59–70, 2004.

[54] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-tree: Indexing XML data for efficient structural joins. In *ICDE*, pages 253–263, 2003.

[55] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig join on indexed XML document. In *VLDB*. ACM, 2003.

[56] K. Kailing, H. P. Kriegel, S. Schönauer, and T. Seidl. Efficient similarity search for hierarchical data in large databases. In *EDBT*, pages 676–693, March. 2004.

[57] Juha Kärkkäinen. Computing the threshold for q-gram filters. In *SWAT*, pages 348–357, 2002.

[58] Raghav Kaushik, Jeffery F Naughton, Philip Bohannon, and Henry F Korth. Covering indexes for branching path queries. In *Proc. of the 2002 ACM SIGMOD international conference on Management of data*, pages 133–144, 2002.

[59] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. ACM SIGMOD Conf.*, page 364, Atlantic City, NJ, May 1990.

[60] Philip N. Klein. Computing the edit-distance between unrooted ordered trees. In *ESA: Annual European Symposium on Algorithms*, 1998.

[61] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley Pub Co, 1997.

[62] Bioinformatic Sequence Markup Language. `http://www.labbook.com/products/standards.asp`, 2001.

[63] Chemical Makeup Language. `http://www.xml-cml.org/information/position.html`, 2001.

[64] Gene Expression Markup Language. `http://xml.coverpages.org/omgGeneExpression.html`, 2000.

[65] Yonk Kyu Lee, Seong-Joon Yoo, and Kyoungro Yoon. Index structures for structured documents. In *ACM First International Conference on Digital Libraries*, pages 91–99, Bethesda, Maryland, USA, Mar 1996.

[66] Changqing Li, Tok Wang Ling, and Min Hu. Efficient processing of updates in dynamic XML data. In *ICDE*, page 13, 2006.

[67] Hanyu Li, Mong-Li Lee, Wynne Hsu, and Chao Chen. An evaluation of XML indexes for structural join. *SIGMOD Record*, 33(3):28–33, 2004.

[68] Hanyu Li, Mong-Li Lee, Wynne Hsu, and Gao Cong. An estimation system for XPath expressions. In *ICDE*, page 54, 2006.

[69] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *The VLDB Journal*, pages 361–370, 2001.

[70] Hartmut Liefke and Dan Suciu. Xmill: an efficient compressor for xml data. In *SIGMOD*, pages 153–164, 2000.

[71] Lipyeow Lim, Min Wang, Sriram Padmanabhan, Jeffrey Scott Vitter, and Ronald Parr. XPathLearner: An on-line self-tuning markov histogram for XML path selectivity estimation. In *VLDB*, pages 442–453, 2002.

[72] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos. The TV-Tree: An index structure for high-dimensional data. *VLDB Journal: Very Large Data Bases*, 3(4):517–542, 1994.

[73] Jiaheng Lu, Ting Chen, and Tok Wang Ling. Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In *CIKM*, pages 533–542, 2004.

[74] Jiaheng Lu, Tok Wang Ling, Chee Yong Chan, and Ting Chen. From region encoding to extended dewey: On efficient processing of XML twig pattern matching. In *VLDB*, pages 193–204. ACM, 2005.

[75] Jiaheng Lu, Tok Wang Ling, Tian Yu, Changqing Li, and Wei Ni. Efficient processing of ordered XML twig pattern. In *DEXA*, pages 300–309, 2005.

[76] Jiaheng Lu, Rui Yang, TokWang Ling, and Anthony K. H. Tung. Efficiently mining frequent trees in a forest. In *In Technical Report, NUS*, 2006.

[77] Nikos Mamoulis, David W. Cheung, and Wang Lian. Similarity search in sets and categorical data using the signature tree. In *ICDE*, pages 75–86, 2003.

[78] J. Mchugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD record*, 26:54–66, Sep 1997.

[79] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*, pages 315–326, September 1999.

[80] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT: 7th International Conference on Database Theory*, 1999.

[81] NCBI Molecular Biology Data Model. `http://www.ncbi.nlm.nih.gov/Sitemap/Summary/asn1.html`, 2002.

[82] Molecular Dynamics Language Home Page (MoDL). `http://violet.csa.iisc.ernet.in/~modl/`, 1999.

[83] Alexandros Nanopoulos and Yannis Manolopoulos. Efficient similarity search for market basket data. *The VLDB Journal*, 11(2):138–152, 2002.

[84] A. Nierman and H.V.Jagadish. Evaluating structural similarity in XML documents. In *Proc. Fifth Int'l Workshop Web and Databases*, June. 2002.

[85] News Industry Text Format (NIFT). `http://www.nitf.org`, 1998.

[86] Naomi Nishimura, Prabhakar Ragde, and Dimitrios M. Thilikos. Finding smallest supertrees under minor containment. *IJFCS: International Journal of Foundations of Computer Science*, 11, 2000.

[87] Open Financial Exchange (OFE). `http://www.ofx.net/ofx/specview/SpecView.html`, 1999.

[88] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-friendly XML node labels. In *SIGMOD Conference*, pages 903–908, 2004.

[89] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *ICDE*.

[90] Neoklis Polyzotis and Minos Garofalakis. Statistical synopses for graph-structured XML databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 3–6, 2002, Madison, WI, USA*, pages 358–369, 2002.

[91] Sven Puhlmann, Melanie Weis, and Felix Naumann. XML duplicate detection using sorted neighborhoods. In *EDBT*, 2006.

[92] Praveen Rao and Bongki Moon. PRIX: Indexing and querying XML using prüfer sequences. In *ICDE*, pages 288–300, 2004.

[93] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The A-tree: An index structure for high-dimensional spaces using relative approximation. In *VLDB*, pages 516–526, 2000.

[94] Torsten Schlieder and Felix Naumann. Approximate tree embedding for querying XML data. In *ACM SIGIR Workshop On XML and Information Retrieval*, Athens, Greece, Jul 2000.

[95] Thomas Seidl and Hans-Peter Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD*, pages 154–165.

[96] Stanley M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6:184–186, December 1977.

[97] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-Tree: A dynamic index for multi-dimensional objects. In *VLDB*, pages 507–518, 1987.

[98] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. Dewitt, and J. Naughton. Relational database for querying XML documents: Limitations and opportunities. In *Proc. 25th Int'l Conf. Very Large Data Bases*, pages 302–314, 1999.

[99] Dennis Shasha and Kaizhong Zhang. *Approximate Tree Pattern Matching*. Oxford University, 1997.

[100] T. Shimura, M. Yoshikawa, and S. Uemura. Storage and retrieval of xml documents using object-relational databases. In *Proc. 10th Int'l Conf. Database and Expert Systems Applications*, pages 206–217, 1999.

[101] Adam Silberstein, Hao He, Ke Yi, and Jun Yang. BOXes: Efficient maintenance of order-based labeling for dynamic XML data. In *ICDE*, pages 285–296. IEEE Computer Society, 2005.

[102] Erkki Sutinen and Jorma Tarhio. On using q-gram locations in approximate string matching. In *Proc. of 3rd Annual European Symposium*, pages 327–340, 1995.

[103] The Niagara System. University of wisconsin, `http://www.cs.wisc.edu/niagara/`.

[104] The Tukwila System. University of washington, `http://data.cs.washington.edu/integration/tukwila/`.

[105] Jiang Tao, Lusheng Wang, and Kaizhong Zhang. Alignment of trees - an alternative to tree edit. In *Theoretical Computer Science (TCS)*, volume 143, pages 75–86, 1995.

[106] J. Tarhio and E. Ukkonen. Boyer-moore approach to apprximate string matching. In *Proc. 2nd Scand. Workshop on Algorithm Theory (SWAT'90), Lecture Notes in Computer Science*, pages 348–359, 1990.

[107] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD Conference*, pages 204–215, 2002.

[108] Pankaj M. Tolani and Jayant R. Haritsa. XGRIND: A query-friendly xml compressor. In *ICDE*, 2002.

[109] TreeBank. University of washington xml repository, `http://www.cs.washington.edu/research/xmldatasets/`.

[110] Esko Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, 92:191–211, 1992.

[111] Haixun Wang and Xiaofeng Meng. On the sequencing of tree structures for XML indexing. In *ICDE*, pages 372–383, 2005.

[112] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. ViST: A dynamic index method for querying XML data by tree structures. In *SIGMOD Conference*, pages 110–121, 2003.

[113] Yuan Wang, David J. DeWitt, and Jin yi Cai. X-diff: An effective change detection algorithm for XML documents. In *Proceedings of the 19th International Conference on Data Engineering*, pages 519–530, Bangalore, India, 2003.

[114] R. Weber, H.-J. Schek, and S. Blott. A quantitative ananlysis and performance study for similarity search methods in high-dimensional space. In *VLDB*, pages 194–205, 1998.

[115] Melanie Weis and Felix Naumann. DogmatiX tracks down duplicates in XML. In *SIGMOD Conference*, pages 431–442. ACM, 2005.

[116] Xiaodong Wu, Mong-Li Lee, and Wynne Hsu. A prime number labeling scheme for dynamic ordered XML trees. In *ICDE*, pages 66–78, 2004.

[117] Zhaohui Xie and Jiawei Han. Join index hierarchies for supporting efficient navigations in object-oriented databases. In *VLDB*.

[118] Rui Yang, Panos Kalnis, and Anthony K. H. Tung. Similarity evaluation on tree-structured data. In *SIGMOD Conference*, pages 754–765. ACM, 2005.

[119] Cui Yu. *High-dimensional Indexing*. PhD thesis, National University of Singapore, Singapore, 2001.

[120] Tian Yu, Tok Wang Ling, and Jiaheng Lu. Twigstacklist¬: A holistic twig join algorithm for twig query with not-predicates on XML data. In *DASFAA*, pages 249–263, 2006.

[121] Yirong Yang Yun Chi, Yi Xia and Richard R. Muntz. Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *IEEE Trans. Knowl. Data Eng.*, 17(2):190–202, 2005.

[122] Mohammed J. Zaki. Efficiently mining frequent trees in a forest. In *SIGKDD*, pages 71–80, 2002.

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

[124] Kaizhong Zhang. Algorithms for the constrained editing distance between ordered labeled trees and related problems. In *Pattern Regonition*, volume 28, pages 463–474, 1995.

[125] Kaizhong Zhang and Danis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SICOMP: SIAM Journal on Computing*, 18:1245–1262, 1989.

[126] Kaizhong Zhang, Dennis Shasha, and Jason T. L. Wang. Approximate tree matching in the presence of variable length don't cares. volume 16, pages 33–66, 1994.