THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Lower Kent Ridge Road, Singapore 119260

TRA2/04

Efficient Processing of XML Pattern Matching: A String Matching-based Approach

Jiaheng LU and Tok Wang LING

February 2004

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

JAFFAR, Joxan
Dean of School

Efficient Processing of XML Pattern Matching: A String Matching-based Approach

Jiaheng Lu, Tok Wang Ling

School of Computing National University of Singapore , Singapore

{lujiahen,lingtw}@comp.nus.edu.sg

Abstract. In this paper, we propose a new approach of indexing XML documents and processing twig patterns in an XML database. Every XML document in the database is labeled with a variation of Dewey ID labeling scheme, namely *Extended Dewey ID*. The unique feature of this labeling scheme is that from the label of an element alone, we can use *finite state transducers* (FST) to derive the names of elements along the path from the root to this element. This feature enables us to directly reduce XML path pattern matching into string matching. We then develop a holistic twig join algorithm, called *TwigComPath*. The algorithm is quite different from the previous strategies in that it solves XML pattern matching problem by string matching and comparing instead of binary relationship matching and stitching. Furthermore, our algorithm only needs to visit the labels of elements that satisfy the leaf node predicates in a twig (or path) pattern; hence, it has performance advantages over the methods that need to visit the labels of all nodes in the pattern. Finally, we provide experimental results to demonstrate the performance benefits of our proposed approaches.

1 Introduction

The preparation of manuscripts which are to be reproduced by photo-offset requires special care. Papers submitted in a technically unsuitable form will be returned for retyping, or canceled if the volume cannot otherwise be finished on time.

XML employs a tree-structured model for representing data. In most of the XML query languages (e.g. Xquery[1],Quilt[2],XPath[3]), structures of XML documents are expressed by twig(or path) patterns, while values of XML elements are used as part of selection predicates. For example, a query pattern by XPath syntax:

bib/book[title="XMLdatabase"]/author[first][last]

This query returns books that (i) are children of bib element (ii) have a child title="XMLdatabase" (iii) have a child author that has two children first and last. This XPath is represented as a twig pattern with elements and string values as nodes.

Finding all occurrences of a twig pattern in a database is a core operation in XML query processing, both in relational implementations of XML databases, and in native XML databases. In the past few years, there have been two main strategies for processing twig pattern queries, namely approaches based on structural index and labeling schemes (labeling schemes are also called numbering schemes or encoding methods in other papers [4-7])

The first approach based on the structural index facilitates traversing through the hierarchy of XML documents by referencing the structural information of documents. (e.g. Dataguides [20], 1-indexes [18], 2-indexes [18], Index Fabric[16] F&B index[8], APEX [21],). While these structural summaries can help reduce the search space for processing structural queries, none of them is a covering–index[8] in that these structural indexes alone only can answer part (not all) of XML queries without consulting the base data.

The second approach is based on a form of labeling scheme that encodes each element by its positional information within the hierarchy of an XML document. In order to answer a query twig pattern, this approach (i) first designs an appropriate labeling scheme to index each document in an XML database, and then (ii) use these labels to perform structural joins without traversing the original XML documents.

For solving the first sub-problem of designing a labeling scheme, most of methods reported in the literature are designed by a tree-traversal order(e.g. pre-and-postorder[12], extended preorder[11], multilevel recursive UID [14]) or textual positions of start and end tags (e.g. containment property[19]) or path expression(e.g. Dewy ID[7]). By using these labeling schemes, one can determine the relationships (e.g. ancestor-descendants) between two elements of the XML database from their labels alone. Further, the order among siblings also can be identified easily.

For solving the second sub-problem of performing structural joins efficiently, several structural join algorithms [4,5,6,11,13] have been developed to take advantage of this extraordinary opportunity to efficiently process twig queries. In particular, Al-Khalifa et al.[13] propose to decompose the twig pattern into many binary relationships, then use Tree-merge or Stack-tree algorithm to match the binary relationships, and finally, "stitch" together basic matches to get the final results. The limitation of this approach is that there is a large size of intermediate results, even when the input and output sizes are more manageable. More recently, Bruno et al. [5] proposes TwigStack algorithm. They use a chain of linked stacks to compactly represent partial results to root-leaf query paths, which are then composed to obtain matches for the twig pattern. They show

that their algorithms are CPU and I/O optimal for query twig patterns that include only ancestor-descendant edges.

In this paper, with an effort to further optimize twig queries processing, we propose a new approach of labeling XML documents and finding twig patterns in an XML database. By using DTD constraints, we design the *extended Dewey ID* labeling scheme, where the label of any element can be used to derive the names of elements along the path from the root to it. We also propose a holistic twig join algorithm. Our idea is to perform twig pattern matching by using string matching and comparing.

The main contributions of this paper are as follows.

- We propose the idea of extending the *Dewey ID* labeling scheme to the *extended Dewey ID* labeling scheme, which enables us to directly reduce XML path pattern matching into *string matching*.
- Based on the *extended Dewey ID*, we develop an algorithm, namely *BranchComPath*, to match a branch pattern, which contains only one node whose fan-out is greater than one. We analytically show that when the query branch pattern has only *ancestor-descendant* relationships in the *fan-out* (not all) edges (the definition of fan-out edges is in Section 5.1), the I/O cost of our algorithm is only proportional to the sum of sizes of the input and the final output.
- We then develop a holistic twig join algorithm, called *TwigComPath*. This algorithm(i) decomposes the twig pattern into several branch patterns, (ii) and uses the bottom–up approach to match each branch pattern to get the final results. We analyze *TwigComPath* to show that its worst-case I/O complexity is linear in the sum of the input and output sizes when the *fan-out* edges of each branch in the twig contain only ancestor-descendant relationships. Furthermore, since *TwigComPath* only needs to visit the labels of leaf nodes in the query, the presence of wildcards "*" in non-leaf nodes does *not* add extra overhead.
- Finally, we show how to use B^+ trees along with *BranchComPath* and *TwigComPath* to further enhance the performance of branch and twig pattern matching.

The rest of the paper proceeds as follows. Section 2 is dedicated to the related work and our motivation. Section 3 gives some preliminary knowledge. We describe the extended Dewey ID labeling scheme in Section 4. Section 5 and 6 present the branch and twig pattern join algorithms respectively. Section 7 discusses the use of B+ tree. We report the experimental results in Section 8, and conclude this paper in Section 9.

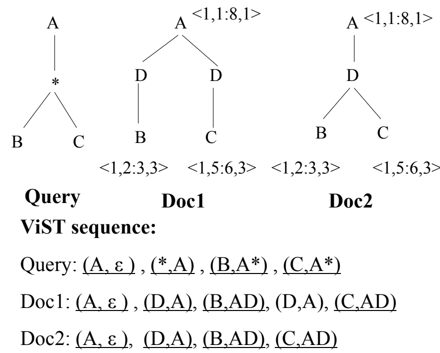


Figure 1. Limitation of TwigStack and ViST

2 Related work

Recently, much research has been focused on how to efficiently perform twig pattern matching on XML documents. Below we will briefly describe two of the-state-of-art contributions made for XML pattern matching, followed by our motivation.

TwigStack Bruno et al[5] propose a holistic twig join algorithm, called *TwigStack.* They use the containment labeling scheme[19], where the positions of XML elements and string values are presented by 4-tuple <*DocId*, *LeftPos*: *RightPos*, *LevelNum*). They show that their algorithms are CPU and I/O optimal for query twig patterns that include only *ancestor-descendant* edges. Also, they propose to use a variation of B-tree, called *XB tree*, to skip reading some data and speed up the process.

However, one of the limitations of their approach is that Algorithm *TwigStack is only optimal* for twig patterns with ancestor-descendant relationships. This algorithm suffers from the sub-optimality for the presence of any parent-child relationhsip in the query. Another limitation of *TwigStack* is that this algrithm is not quite efficient for processing queries with withwild "*", which can match any single element in an XML document. Consider a query pattern and two documents in Figure 1. From the labels of A(1,1:8,1), B(1,2:3,3), C(1,5:6,3) alone, we cannot identify which document indeed matches this query. To answer this query, one has to visit all the labels of elements that likely match wildcard "*". So this operation increases I/O cost of their algorithm.

ViST Wang et al[4] have proposed a novel method called ViST that transforms the XML data trees and twig patterns into the *structure-encoded sequences*. A structure-encoded sequences is a two dimensional sequence of (*symbol*, *prefix*) pairs. ViST performs subsequence matching on the

structure-encoded sequences to find twig patterns in XML documents. These sequences are stored in a virtual suffix tree.

One obvious limitation of ViST is that this approach may get false matchings. Figure 1 illustrates such a case. The structure-encoded sequence of the query twig query is a subsequence of the structure-encoded sequence of Doc1 and Doc2. However, the twig pattern query occurs only in Doc2, and the match detected in Doc1 is a false matching. Another limitation of ViST is that this method is not quite efficient for processing queries with wildcard "//". Since this wildcard can match all the children or desecdants of symbol, ViST should search all (symbol,//) keys to avoid missing solutions. This search is uauslly expensive.

Our Motivation The motivations of this paper include (1) to develop a twig pattern matching algorithm that has better I/O property than *TwigSatck* for the query patterns with *parent-child* relationships (2) to ensure that this algorithm correctly returns all answers (3) to enable this algorithm to answer the queries with wildcards "//" and "*" as efficiently as those without wildcards.

3 Preliminaries

3.1 Data model

An XML document is usually modeled as a tree structure. Each node in a tree corresponds to an element or a value. Each node can have a list of (*attribute*, *value*) pairs associated with it. An attribute is usually represented as a sub-element of an element. Values are represented by character data (CDATA, PCDATA) and occur at the leaf nodes. The tree edges present a relationship between two elements or between an element and a value.

Queries in XML query languages make use of twig patterns for matching relevant portions of data in an XML database. The twig pattern node may be an element tag, wildcard(*)symbol and string values. The query twig pattern edges are either parent-child edges(indicated by single lines) or ancestor-descendant edges (indicated by double lines).

3.2 Twig pattern matching

Given a twig pattern Q and an XML database D, a match of Q in D is identified by a mapping from nodes in Q to elements in D, such that: (i) query node predicates are satisfied by the corresponding database elements (wherein, wildcard "*" matches any single tag) and (ii) the structural

(parent-child and ancestor-descendant) relationships between query nodes are satisfied by the corresponding database elements. The answer to query Q with *n* nodes can be represented as a list of *n-ary* tuples, where each tuple $(d_1,\ldots d_n)$ consists of the database elements that identify a distinct match of Q in D. In the rest of this paper, for presentation clarity, “node” refers to a node in a twig pattern, while “element” refers to an element in an XML database.

3.3 Dewey ID labeling scheme

In *Dewey ID* labeling scheme[7], the position of an element in an XML database is presented by a binary tuple (*DocId*, *DeweyID*), where *DocId* is the identifier of the document and *DeweyID* is decided as follows: (i) *root* is labeled by an empty string; (ii) for a non-root element *u*, *DeweyID(u)= DeweyID(s).i*, where *u* is the *i-th* child of *s*. Each label can be efficiently generated by a *depth-first* traversal of trees. We refer to this labeling scheme as the *simple Dewey ID* in this paper.

Simple Dewey ID has a nice property: one can derive the ancestors of an element from its label alone. For example, if an element *u* is labeled with “1.2.3”, then the parent of *u* is labeled with “1.2” and its grandparent is labeled with “1”. With the knowledge of this property, we further consider that if the names of ancestors of an element *u* can be derived from *DeweyID(u)* alone, then *XML path pattern matching* can be reduced to *string matching*. For example, if we know that the label “1.2.3” of an element *u* presents the path “a/b/c” from *root* to *u*, then it is quite straightforward to identify whether the path of *u* matches a path pattern (e.g. “a//c”). Inspired by this observation, we propose an *extended Dewey ID* that extends the *simple Dewey ID* and provides a possibility for us to design a new approach to match twig patterns.

4 Extended Dewey ID

4.1 Preliminary definitions

XML documents may be in accordance to a schematic representation described by a DTD(Document Type Definition). DTDs are significant for XML data, because schematic information is essential at all levels of database design, implementation and usage. DTDs can have multiple uses in querying XML data[22] and creating XML views[17]. To the best of our knowledge, this is the first paper to use DTDs to design a new labeling scheme for efficient processing of XML pattern matching.

DEFINITION 4.1 (DTD) *A DTD over a finite alphabet Σ consists of a root type in Σ and a set of rules with the form s→R(s) where s∈Σ is a start symbol and R(s) is a regular expression over Σ.*

```
<!ELEMENT bib  (book*)>
<!ELEMENT book ((author | editor)+, title, chapter* )>
<!ELEMENT author  (last, first )>
<!ELEMENT editor  (last, first, affiliation* )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT last  (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT chapter (title, section*)>
<!ELEMENT section (title, section*)>
<!ELEMENT title (#PCDATA)>
```

Figure 2 A sample DTD

Notation In the subsequent discussion, we shall frequently use these notations: given a DTD rule: *s→R(s),* let |R(s)| denote the number of *distinct* tags in R(s) and t_k denote the k-th *distinct* tag of R(s). For example, given a DTD rule:

$$s \rightarrow (a|b)^{*}a\cdot(cd)^{+},$$

where |R(s)|=4, t_1=a, t_2=b, t_3=c, t_4=d.

Further, Let N denote the natural number set and $\Re$ denote the regular expression set over the finite alphabet Σ. □

Recall that our goal is to design a labeling scheme, where given an element *u*, we can derive the tags of ancestors of *u* from *label(u)* alone. In order to achieve this purpose, we need a function *f(x,R(s))*, given an integer *x* and a DTD rule *R(s)*, *f(x,R(s))* maps *x* and *R(s)* into the tag t∈R(s).

DEFINITION 4.2 *A function f(x,R(s)):* $N \times \Re \rightarrow \Sigma$ *can be defined by f(x,R(s))=* t_k*, where* t_k *is the k-th distinct tag in R(s).*

$$k = \begin{cases} |R(s)| & \text{if } x \bmod |R(s)| = 0 \\ x \bmod |R(s)| & \text{otherwise} \end{cases}$$

EXAMPLE 4.2 Consider the DTD rule: $s \rightarrow (a|b)^{*}a\cdot(cd)^{+}$.

By this function,

$$f(x, (a|b)^{*}a\cdot(cd)^{+}) = \begin{cases} a & \text{if } x \bmod 4=1 \\ b & \text{if } x \bmod 4=2 \\ c & \text{if } x \bmod 4=3 \\ d & \text{if } x \bmod 4=0 \end{cases}$$

Note that this function builds up the mapping between the natural number and the tag name in a DTD regular expression.

4.2 Extended Dewey ID and FST

With the above introduction, we proceed to formulate two converse procedures. First, in the *extended Dewey ID* labeling scheme, for any element *u*, we use a formula to *encode* the names of elements in the path from the root to *u* into the label of *u*. Second, in order to answer the XML queries, we use a *finite state transducer (FST)* to *decode* these elements names from the label of *u*. In both procedures, we make the crucial use of the function *f(x,R(s))*.

4.2.1 Extended Dewey ID

As its name suggests, the *extended Dewey ID* labeling scheme is a variation of *simple Dewey ID*. The label of each element also can be effectively generated by a *depth-first* traversal of the tree. Each label is still presented as a binary tuple (*DocId*, *ExDeweyID*), where *DocId* is the identifier of the document, and for each element *u*, *ExDeweyID(u)* is equal to *ExDeweyID(s).x* , where *s* is the parent of *u*. The computation method of *x* is a little more involved than *simple Dewey ID.* With the *extended Dewey ID*, for any element *u*, assume that the tag of *u* is the k-th distinct tag in its corresponding DTD rule *s*→R(s).

Rule(1):if u is the string value of an element , then x=0;

Rule(2): otherwise, assume that the tag of u is the k-th distinct tag in its corresponding DTD rule s→R(s).

(2.1) if u is the first child of s, then x=k;

(2.2) otherwise assume that the last component of the label for the left sibling of u is y, then

$$x = \begin{cases} \left\lfloor \frac{y}{|R(s)|} \right\rfloor \cdot |R(s)| + k & w < k \quad \text{(i)} \\ \left\lceil \frac{y}{|R(s)|} \right\rceil \cdot |R(s)| + k & w \geq k \quad \text{(ii)} \end{cases}$$

where， if (y mod |R(s)|)=0, then w = |R(s)| , else w = y mod |R(s)|.

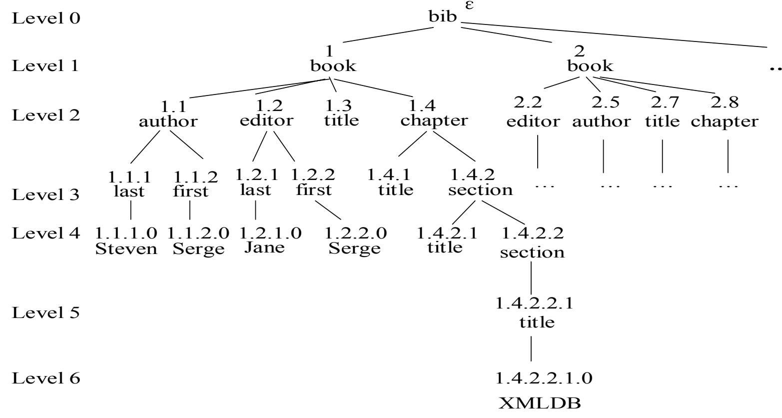


Figure 3 An XML tree with extended Dewey ID

EXAMPLE 4.3 *Figure 3 illustrates an XML document tree that conforms to the DTD in Figure 2 and the* ExDeweyID *label of each element (for clarity,* DocId *of each element is omitted here). For instance, the label of "author" under "book(2)" is computed as follows. Here k=1("author" is the first tag in its DTD rule), y=2(the last component of 2.2 is 2), |R(s)|=4, so w=y mod |R(s)|=2. Thus w≥k, and by Equation (ii), x=5. So the "author" is assigned "2.5".* □

In order to support the dynamic updates of XML data, similar to *simple Dewey ID*, we may reserve extra spaces to accommodate the future insertions. When the reserved spaces are consumed, only the following *siblings* and *descendants* of elements may need to be relabeled[7]. Further, note that *extended Dewey ID* also allows for *checking order* (e.g. in Figure 3, the element "*author(2.5)*" follows the element "*editor(2.2)*", for 5>2)

The following lemma compares the space complexity of *extended Dewey ID* with *simple Dewey ID*.

LEMMA 4.1 *The extended Dewey ID labeling scheme does not asymptotically change the index size complexity of the simple Dewey ID.*

PROOF: *We first prove that given a DTD rule s→R(s), the gap between the values of the last components of labels for every two neighboring elements under s is no more than |R(s)|. We prove this statement in two cases:*

Case(1): One of the two elements is the first child. According to Rule (2.1) and (2.2), the gap g between the values of the last component of labels is:

$$g = \left\lfloor \frac{y}{|R(s)|} \right\rfloor \cdot |R(s)| \;\; or \;\; g = \left\lceil \frac{y}{|R(s)|} \right\rceil \cdot |R(s)|$$

where y=k≤|R(s)|. Thus, g is no more than |R(s)|.

Case(2): None of the two elements is the first child.

Case (2.1) When w<k, according to the equation (i) of Rule (2.2), the gap g between the values of the last components of labels is:

$$g = \left\lfloor \frac{y}{|R(s)|} \right\rfloor \cdot |R(s)| + k - y \le y + k - y = k$$

Since $k \leqslant |R(s)|$, *it is easy to see that* $g \leqslant |R(s)|$.

Case (2.2) When $w \geqslant k$,

Case (2.2.1) if (y mod |R(s)|)=0, then w=|R(s)|.

$$g = \left\lfloor \frac{y}{|R(s)|} \right\rfloor \cdot |R(s)| + k - y = y + k - y = k$$

Since $k \leqslant |R(s)|$, $g \leqslant |R(s)|$.

Case (2.2.2) if (y mod |R(s)|) $\neq 0$, *then w=y mod |R(s)|.*

$$g = \left\lfloor \frac{y}{|R(s)|} \right\rfloor \cdot |R(s)| + k - y = |R(s)| - w + y + k - y = |R(s)| - w + k$$

Since $w \geqslant k$, $g = |R(s)| - w + k \leqslant |R(s)|$.

By the discussion in the above two cases, we conclude that given a DTD rule s→R(s), the gap between the values of the last components of labels for every two neighboring elements under s is no more than |R(s)|. Further, given an element u, the value of each component i in the label of u in the extended Dewey ID is no more than the product of $|R(s_i)|$ *and the value of the corresponding component in the simple Dewey ID, where* $R(s_i)$ *is the DTD rule defined on component i. Thus, with the binary representation of labels, the length of each component in the extended Dewey ID is at most* $\log_2|R(s_i)|$ *more than that of the simple Dewey ID. Since* $|R(s_i)|$ *is usually small, it is quite reasonable to consider* $\log_2|R(s_i)|$ *to be a small constant.*

As a result, the extended Dewey ID labeling scheme does not asymptotically change the index size complexity of the simple Dewey ID. □

4.2.2 Finite state transducer

Given the label of an element *u*, we can construct a *finite state transducer (FST)* to print the name of elements along the path from the *root* to *u*.

DEFINITION 4.3 (Finite State Transducer) *Given an XML document D and a DTD against D, we can construct a* finite state transducer *(*FST*) to translate a sequence of numbers into a sequence of tag names. FST is a 5-tuple* (I, S, i, δ, o), *where*

(i) the input set $I = N \cup \{0\}$;

(ii) the set of states $S = \Sigma \cup \{PCDATA\}$, *where* Σ *is the finite alphabet of DTD and* PCDATA *is a state indicating that the current input presents the string value of an element;*

(iii) the initial state i *is the root of DTD ;*

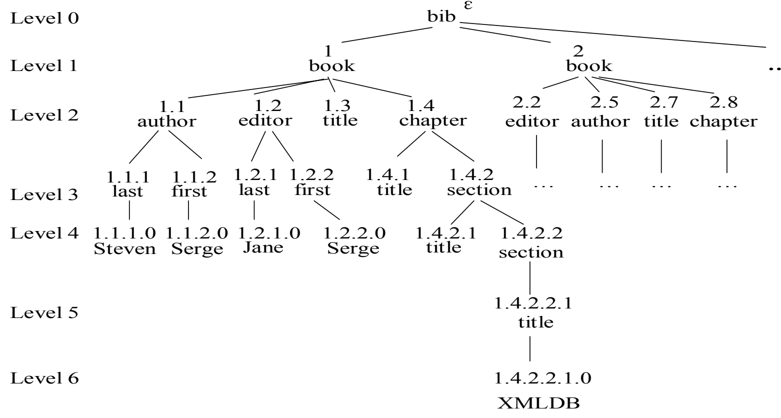


Figure 3 An XML tree with extended Dewey ID

EXAMPLE 4.3 *Figure 3 illustrates an XML document tree that conforms to the DTD in Figure 2 and the* ExDeweyID *label of each element (for clarity,* DocId *of each element is omitted here). For instance, the label of "author" under "book(2)" is computed as follows. Here k=1("author" is the first tag in its DTD rule), y=2(the last component of 2.2 is 2), |R(s)|=4, so w=y mod |R(s)|=2. Thus w≥k, and by Equation (ii), x=5. So the "author" is assigned "2.5".* □

In order to support the dynamic updates of XML data, similar to *simple Dewey ID*, we may reserve extra spaces to accommodate the future insertions. When the reserved spaces are consumed, only the following *siblings* and *descendants* of elements may need to be relabeled[7]. Further, note that *extended Dewey ID* also allows for *checking order* (e.g. in Figure 3, the element "*author(2.5)*" follows the element "*editor(2.2)*", for 5>2)

The following lemma compares the space complexity of *extended Dewey ID* with *simple Dewey ID*.

LEMMA 4.1 *The extended Dewey ID labeling scheme does not asymptotically change the index size complexity of the simple Dewey ID.*

PROOF: *We first prove that given a DTD rule s→R(s), the gap between the values of the last components of labels for every two neighboring elements under s is no more than |R(s)|. We prove this statement in two cases:*

Case(1): One of the two elements is the first child. According to Rule (2.1) and (2.2), the gap g between the values of the last component of labels is:

$$g = \left\lfloor \frac{y}{|R(s)|} \right\rfloor \cdot |R(s)| \ \ or \ \ g = \left\lceil \frac{y}{|R(s)|} \right\rceil \cdot |R(s)|$$

where y=k≤|R(s)|. Thus, g is no more than |R(s)|.

Case(2): None of the two elements is the first child.

Case (2.1) When w<k, according to the equation (i) of Rule (2.2), the gap g between the values of the last components of labels is:

$$g = \left\lfloor \frac{y}{|R(s)|} \right\rfloor \cdot |R(s)| + k - y \le y + k - y = k$$

Since $k \leqslant |R(s)|$*, it is easy to see that* $g \leqslant |R(s)|$.

Case (2.2) When $w \geqslant k$,

Case (2.2.1) if (y mod |R(s)|)=0, then w=|R(s)|.

$$g = \left\lfloor \frac{y}{|R(s)|} \right\rfloor \cdot |R(s)| + k - y = y + k - y = k$$

Since $k \leqslant |R(s)|$, $g \leqslant |R(s)|$.

Case (2.2.2) if (y mod |R(s)|) $\neq 0$*, then w=y mod |R(s)|.*

$$g = \left\lfloor \frac{y}{|R(s)|} \right\rfloor \cdot |R(s)| + k - y = |R(s)| - w + y + k - y = |R(s)| - w + k$$

Since $w \geqslant k$, $g = |R(s)| - w + k \leqslant |R(s)|$.

By the discussion in the above two cases, we conclude that given a DTD rule s→R(s), the gap between the values of the last components of labels for every two neighboring elements under s is no more than |R(s)|. Further, given an element u, the value of each component i in the label of u in the extended Dewey ID is no more than the product of $|R(s_i)|$ *and the value of the corresponding component in the simple Dewey ID, where* $R(s_i)$ *is the DTD rule defined on component i. Thus, with the binary representation of labels, the length of each component in the extended Dewey ID is at most* $\log_2|R(s_i)|$ *more than that of the simple Dewey ID. Since* $|R(s_i)|$ *is usually small, it is quite reasonable to consider* $\log_2|R(s_i)|$ *to be a small constant.*

As a result, the extended Dewey ID labeling scheme does not asymptotically change the index size complexity of the simple Dewey ID. □

4.2.2 Finite state transducer

Given the label of an element *u*, we can construct a *finite state transducer (FST)* to print the name of elements along the path from the *root* to *u*.

DEFINITION 4.3 (Finite State Transducer) *Given an XML document D and a DTD against D, we can construct a* finite state transducer *(*FST*) to translate a sequence of numbers into a sequence of tag names. FST is a 5-tuple* (I, S, i, δ, o)*, where*

(i) the input set I =N ∪ {0};

(ii) the set of states S= Σ ∪ {PCDATA}*, where* Σ *is the finite alphabet of DTD and* PCDATA *is a state indicating that the current input presents the string value of an element;*

(iii) the initial state i *is the root of DTD ;*

It is important to note that the I/O cost of our approach is usually much smaller than that of Algorithm *PathStack*[5], for our method only needs to scan the labels of query leaf nodes but *PathStack* needs to scan the labels of *all* nodes.

5.2 Path pattern matching algorithm

5.2.1 *Definition and notation*

In a query branch pattern, there exists only one node B, called *branchpoint*, whose fan-out is greater than one. The edges directly connecting to the branchpoint in different branches are called *fan-out* edges. A branch pattern with *n* branches has *n fan-out* edges. Moreover, let β denote a branch pattern and q_i denote the i-th leaf node in β, and L_B denotes the path pattern from the root to the branchpoint of β (see Fig 5). Finally, let L_{qi} denote the path pattern from the root to the leaf node q_i.

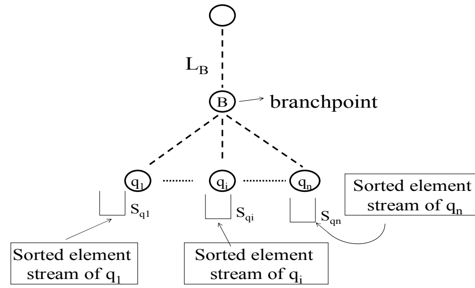
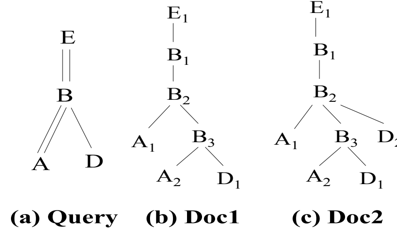


Figure 5 A branch pattern and data streams

We assume that there is a data stream S_{qi} associated with the *leaf* node q_i in pattern β. The elements in each stream S_{qi} are sorted by their (*DocID*, *ExDeweyID*), where *ExDeweyID* is sorted by *lexicography order*. The operations over streams are *eof*, *advance*, *getDocID* and *getED*. The last two operations return the *DocID* and the *ExDeweyID* of the current element, respectively.

As shown in the previous section, it is easy for us to know whether a label satisfies the corresponding path pattern. Thus, the key task of our branch join algorithm is to identify whether a label has a prefix that (i) matches the query root-branchpoint path (i.e. pattern L_B) and (ii) is a common substring for at least one label from each data stream. We address this problem using the following definitions and lemma.

DEFINITION 5.1 (**Matched-Prefix set**) *Given a branch pattern β and an XML database D, for any ExDeweyID label d_i from data stream S_{qi}, if d_i matches the root-leaf path L_{qi} of β, then we define a* Matched-Prefix(MP) *set for d_i,* MP(d_i, β)= {s | (s is a prefix of d_i)∧(s satisfies pattern L_B of β) ∧ (d_i –s satisfies pattern L_{qi} – L_B of β)}



(a) Query (b) Doc1 (c) Doc2

Figure 6 A sample query and two document trees

EXAMPLE 5.1 *Consider the query β and the XML Doc1 in Figure 6(a),(b).*

$MP(E_1B_1B_2A_1, \beta)=\{E_1B_1, E_1B_1B_2\}$,

$MP(E_1B_1B_2B_3A_2, \beta)=\{E_1B_1, E_1B_1B_2, E_1B_1B_2B_3\}$

$MP(E_1B_1B_2B_3D_1, \beta) = \{E_1B_1B_2B_3\}$.

Note that E_1B_1 and $E_1B_1B_2$ are not in $MP(E_1B_1B_2B_3D_1, \beta)$, since $B_2B_3D_1$ and B_3D_1 do not match pattern "/D".

DEFINITION 5.2 (**Fixed-end solution**) *Given a string t matching a pattern L, we say that a solution S of t to L is a fixed-end solution iff the last component of S is the last character of t.*

For example: given the string "$A_1A_2B_1B_2$" and pattern "A//B", the *fixed-end* solutions are (A_1,B_2) and (A_2,B_2).

The above two definitions are important for establishing the following lemma.

LEMMA 5.1 *Consider a branch pattern β and an XML database D. Given a label t in D and a string s in MP(t,β), the fixed-end solutions of s to pattern L_B are part of at least one final solution if and only if, for each stream S_{qi} in D, there exists a label d_i in S_{qi} such that $s \in MP(d_i, \beta)$.*

PROOF: [only if part] *In this part, we assume that the fixed-end solutions of s to pattern L_B are part of at least one final solution and try to prove for each stream S_{qi} in D, there exists a label d_i in S_{qi} such that $s \in MP(d_i, \beta)$. This part is easy. According to the definition of MP sets, string s satisfies the pattern L_B, which is a common part of each root-leaf path patterns. Since the fixed-end solutions of s to pattern L_B are part of at least one final solution, for each stream S_{qi} in D, there exists a label d_i in S_{qi} such that $s \in MP(d_i, \beta)$.*

[If part] *Now, we assume that for each stream S_{qi} in D, there exists a label d_i in S_{qi} such that $s \in MP(d_i, \beta)$ and try to prove the fixed-end solutions of s to pattern L_B are part of at least one final solution. The key point for s to be part of at least one final solution is that s should be a common prefix for at least one label d_i from each data stream S_{qi}, such that d_i matches pattern L_{qi} and d_i-s matches pattern L_{qi}. Since for each stream S_{qi} in D, there should be a label d_i in S_{qi} such that $s \in$*

$MP(d_i, β)$, s is a common prefix for each label d_i in S_{qi}. Thus, we may conclude by the definition of MP sets that the fixed-end solutions of s to pattern L_B are part of at least one final solution.

EXAMPLE 5.2 *Consider the branch pattern β and the XML Doc1 in Figure 6(a), (b) again, which we use to explain Lemma 5.1. Since "$E_1B_1B_2B_3$" is in $MP(E_1B_1B_2B_3A_2, β)$ and $MP(E_1B_1B_2B_3D_1, β)$, and the fixed-end solution of "$E_1B_1B_2B_3$" to pattern E//B is (E_1,B_3), by Lemma 5.1, (E_1,B_3) is* guaranteed to be *part of at least one final solution. And it is easy for us to verify that (E_1,B_3) is part of the final solution (E_1,B_3,A_2,D_1)in Figure 6(b).*

Thus, a necessary condition for a label contributing to final answers is that it should have a prefix p, such that p is a common string in the MP sets of at least one label from each data stream. In the branch join algorithm, we cache such prefixes p in a set *ComPath* and use them to check whether any input label could be part of final answers.

Finally, we show the classic definition of lexicography order as follows.

DEFINITION 5.3 (Lexicography order) *Given two strings a and b, a>b by lexicography order, iff*
(1) b is a prefix of a, ***or***
(2) for the first characters at which a and b differ, the character in b is less than that in a.

In this paper, we use lexicography order to compare two extended Dewey ID labels. But an extended Dewey ID labels is a sequence of integers, so the characters of a string in the above definition correspond to the integers in an extended Dewey ID label.
For example, consider Figure 3.
(1) The label *(1.4)* of element "chapter" is less than the label *(1.4.2.2.1)* of element "title" (i.e. *1.4<1.4.2.2.1*), because *1.4* is a prefix of *1.4.2.2.1*.
(2) The label *(2.2)* of element "editor" is less than the label *(2.5)* of element "author" (i.e. *2.2<2.5*), because the first integers at which *2.2* and *2.5* differ are *2* and *5* respectively and the integer *2* in *2.2* is less than the integer *5* in *2.5* (i.e. 2<5).

5.2.2 *Algorithm BranchComPath*

Algorithm *BranchComPath* in Figure 7 computes answers to a query branch pattern. The algorithm operates in two phases. In the first phase (line 1-7), some (but not all) solutions to the indi-

vidual query root-leaf path are output. In the second phase (line 8), these solutions are merge-joined to compute the answers to the branch pattern.

In the first phase, the key idea of *BranchComPath* is (i) to use the set *ComPath* to cache partial and total strings that possibly become the common strings in the MP sets of at least one label from each data stream, and (ii) to output the root-leaf path solutions for the strings that has a prefix in *ComPath* and matches the individual path pattern.

Before we delve into details of the first phase, we shall first introduce two kinds of strings in set *ComPath*. When a string *s* is inserted into *ComPath*, if it is a common one in the MP sets of at least one label from *each* data stream, then *s* is called a ***qualified string***. Otherwise, at the point of insertion, *s* is called a ***candidate string*** if s is a common string in the MP sets of at least one label from *part* (not each) of data streams.

In *BranchComPath*, at *every* point in the computation, every two strings in *ComPath* (including qualified and candidate ones) are guaranteed to be a prefix pair. Before a string *s* is appended into *ComPath*, we should first delete any string that has no prefix relationship with *s*.

We now go through the first phase of *BranchComPath*. Initially, we advance each stream to the first label that matches the individual query root-leaf path (i.e. pattern L_{qi})(line 1). Line 3 advances the streams, where (i)the current label is guaranteed no contribution to any final answer(if $s \neq$prefix(*max*)); or (ii) its path solutions have been output(if $s \in$ComPath). Line 5 inserts the qualified strings into *ComPath*. Line 6 inserts each string in set MP(d_m ,β) into *ComPath* as a candidate string, such that MP(d_m ,β) contains the minimal string in all MP sets of current labels (in this paper, we always compare strings by lexicography order). Finally, line 7 outputs path solutions that are the concatenations of two parts: the fixed-end solutions to pattern L_B and $L_{qi} - L_B$.

The interesting step is in line 4. Only if every two strings in all MP sets are a prefix pair (i.e. $\forall x \in$MP(d_i, β) and $\forall y \in$MP(d_j, β): x(or y) is a prefix of y(or x)), we need to insert qualified or/and candidate strings into *ComPath*. The correctness of this step is showed as follows.

There is a simple but important property about lexicography order.

PROPERTY 5.1 *If there are two strings a and b, such that* a>b *by lexicography order and* b *is not a prefix of* a*, it holds that* a·c>b·d *for any string* c*,* d.

PROOF: *Since a>b and b is not a prefix of a, string a and b can be written by:*

$a = A_1A_2...A_nA_{n+1}...A_{n+i}$; $b = A_1A_2...A_nB_{n+1}...B_{n+m}$;

where $n \geq 0$ *and* $A_{n+1} > B_{n+1}$.

It is easy to see that whatever strings c and d are concatenated with a and b, by lexicography order,

$$A_1A_2...A_nA_{n+1}...A_{n+i} \bullet c > A_1A_2...A_nB_{n+1}...B_{n+m} \bullet d.$$

This concludes the correctness of this propoerty. □

Algorithm BranchComPath (β)

// Assume all elements have the same DocId.

//Phase 1

01 **for** q_i in leafnodes(β) locateMatchingLabel(S_{q_i} , β);

02 **while** (¬end(β)) {

// Let D = { *getED*(S_{q_i}) | q_i ∈ leafnodes(β)}.

03 advanceStreams(β);

04 **if** (∀ d_i, d_j ∈ D, ∀ x ∈ MP(d_i,β), ∀ y ∈ MP(d_j,β) : (x,y) is a prefix pair)

05 {addQualifiedToComPath(β);

06 addCandidateToComPath(β);}//end if

07 outputSolutions (β); }//end while

//phase 2

08 MergeAllPathSolutions( );

Function end(β)

01 **return** ∀ q_i ∈ leafnodes (β) : eof(S_{q_i});

Procedure locateMatchingLabel(S_{q_i} , β)

01 **do** advance(S_{q_i});

02 **while** ((*getED*(S_{q_i}) not match L_{q_i} of β) ∧ ¬eof(S_{q_i}));

Procedure advanceStreams(β)

01 let *max* be the maximal string in $\bigcup_{d_i \in D} MP(d_i, \beta)$ by lexicography order;

02 **for** q_i in leafnodes(β)

03 **if** (∀ s ∈ MP(*getED*(S_{q_i}),β) : s ∈ ComPath ∨ s ≠ prefix(*max*))

04 locateMatchingLabel(S_{q_i} , β);

Procedure addQualifiedToComPath(β)

01 **for** p ∈ $\bigcap_{d_i \in D} MP(d_i, \beta)$ **if** (p ∉ *ComPath*) append p to *ComPath*;

Procedure addCandidateToComPath (β)

//Assume MP(d_m,β) contains the minimal string in $\bigcup_{d_i \in D} MP(d_i, \beta)$;

01 **for** p ∈ MP(d_m,β) **if** (p ∉ *ComPath*) append p to *ComPath*;

Procedure outputSolutions (β)

01 **for** q_i in leafnodes(β)

02 **for** s ∈ (MP(*getED*(S_{q_i}),β) ∩ *ComPath*)

03 **for** fixed-end solution t_B of s to pattern L_B

04 **for** fixed-end solution t_i of ($getED(S_{q_i}) - s$) to pattern $L_{q_i} - L_B$

05 outputWithBlocking (t_B , t_i);

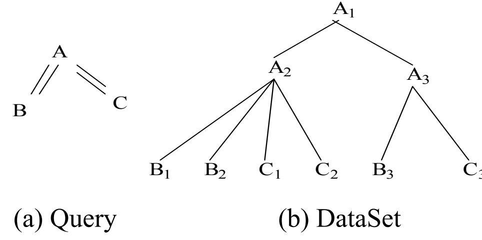
Figure 7 Algorithm BranchComPath

If there exist strings a and b in stream S_A and S_B respectively such that a>b and b≠prefix(a), then we can safely advance the stream S_B without any insertion to *ComPath*. This is because, according to Property 4.1, it holds that a·c>b·d for any string c, d. Thus, *b* cannot become a prefix for any label after a in S_A. So if *b* has not already been inserted into *ComPath, b* is guaranteed not to be part of final solutions. As a result, we can safely advance the stream S_B without any insertion to *ComPath*.

The second phase of Algorithm *BranchComPath* is linear in the sum of the input (the solutions to individual path pattern) and the output (the answers to the query branch pattern) sizes, only when the inputs are in sorted order. This can be achieved by using *blocking* [5,13] to delay some outputs until no answers prior to them can be computed. The details of how to naturally achieve this in *BranchComPath* are discussed in the next section.

EXAMPLE 5.4: *Consider the query* β *and data in Fig 6(a),(b) again. We use them to illustrate Algorithm BranchComPath. First of all, we scan* A_1 *and* D_1*(line 1). The matched-prefix set of* A_1 *is* $\{E_1B_1, E_1B_1B_2\}$ *and that of* D_1 *is* $\{E_1B_1B_2B_3\}$*. Since* E_1B_1 *and* $E_1B_1B_2$ *are prefixes of* $E_1B_1B_2B_3$*, line 3 does not advance any stream and the IF condition in line 4 returns true. Subsequently, there is no common string in both sets, so no string is inserted into ComPath as a qualified string(line 5). In line 6,* MP(d_m, β)= $\{E_1B_1, E_1B_1B_2\}$*,* so *we insert* E_1B_1 *and* $E_1B_1B_2$ *into ComPath. Then, in line 7, we output path solutions* (E_1,B_1,A_1) *and* (E_1,B_2,A_1)*. Subsequently, scan* A_2*(line 3) and insert the qualified string* $E_1B_1B_2B_3$ *into ComPath(line 5), then output* (E_1,B_1,A_2) (E_1,B_2,A_2) (E_1,B_3,A_2) *and* (E_1,B_3,D_1)*(line 7). Finally, in the second phase, six path solutions are merged to form one final result* (E_1,B_3,A_2,D_1)*.*

5.3 Blocking Results



(a) Query (b) DataSet

Figure 8. An example for blocking

As mentioned in the previous section, blocking should be used to ensure that the inputs of the second phase of *BranchComPath* are in the sorted order. Consider the simple branch query and dataset in Figure 8(a) and (b) to. When Algorithm *BranchComPath* scan B_1,C_1 and insert two strings “A_1”, “A_1A_2” into set *ComPath* as qualified strings, we cannot immediately output the corresponding path solutions $<A_1,B_1>,<A_2,B_1>,<A_1,C_1>,<A_2,C_1>$. This is because there remains the possibility of a new element after B_1 and C_1 whose matched-prefix set also contains strings “A_1” or “A_1A_2” so that the output path solutions may not be sorted. Therefore, we now propose a procedure that returns solutions in the sorted order.

For this purpose, we maintain a list associated with each string in ComPath and each input data stream. When a new string *s* is inserted into the set ComPath, for each data stream S_{qi}, we initialize a list for s and S_{qi}, denoted by list(s,i). Each element in list(s,i) is a string which satisfy pattern L_{qi}-L_B . For example, in Figure 8, we maintain four lists.

When B_1 and C_1 are scanned,

List(“A_1”,B)={B_1}, List(“A_1”,C)={C_1}, List(“A_1A_2”,B)={B_1}, List(“A_1A_2”,C)={C_1},

And when B_2 and C_2 are scanned,

List(“A_1”,B)={B_1,B_2}, List(“A_1”,C)={C_1,C_2}, List(“A_1A_2”,B)={B_1,B_2}, List(“A_1A_2”,C)={C_1,C_2}.

The main ideas of the algorithm remain the same, but we do not output solutions immediately when we detect them. Instead, we accumulate the partial solutions in their respective lists in the sorted order. When we finally output solutions, they are guaranteed to be in the right order. In particular, when a new element p is appended into *ComPath*, we delete any string, say s, which is not a prefix-pair with p in *ComPath* and output all binary tuples $<s,d_i>$ for each $d_i \in$ list(s,i).

For example, in Figure 8, when B_3, C_3 are scanned, since “A_1A_2” is not a prefix-pair with “A_1A_3”, we delete “A_1A_2” from *ComPath* and output four binary tuples:

$<$“A_1A_2”, “B_1”$>$,$<$“A_1A_2”, “B_2”$>$,$<$“A_1A_2”, “C_1”$>$,$<$“A_1A_2”, “C_2”$>$.

The following definition and lemma show that the first element *s* of the output binary tuple <s,d_i> is sorted by the *post-order*.

DEFINITION 5.4 (Post-order) *Given two strings a and b, a>b by post-order, iff*

(1) a is a prefix of b, or

(2) a is not a prefix of b and a>b by lexicography order.

LEMMA 5.2 *In Algorithm BranchComPath, the strings deleted from ComPath are sorted by the ascending Post-order.*

PROOF: *We can prove this lemma by contradiction. If there exsit two strings s and p, such that s<p but p is deleted from ComPath earlier than s, then there are two cases:*

(i) *p is a prefix of s. If we delete s and p for the insertion of the same node, then we can first delete s followed by p. This is trivial. Otherwise, s and p are deleted for the insertions of different nodes. But it is impossible, because any string which is inserted between s and p must be a prefix-pair with s and p. Thus, s and p must be deleted together.*

(ii) *p is not a prefix of s and s<p. It is also impossible, because, according to the property of post-order, if p is not a prefix of s , then whatever string s' and p' are appended with s and p, it holds that ss'<pp'. Since all data in input streams are sorted by lexicography order, ss' is prior to pp'. Thus, s is inserted to set ComPath earlier than p. And considering that p is not a prefix of s, it is impossible that p is deleted from ComPath earlier than s.* □

Since the output data are sorted by the post-order, the second phase of Algorithm *BranchComPath* is linear in the sum of its input (the binary-tuple solutions to the individual path pattern) and the output (the answers to the query branch pattern) sizes. The binary-tuple solutions are merged and converted the N-tuple format, where N is the number of nodes in the query branch.

The following example illustrates the blocking procedure proposed in the above.

EXAMPLE 5.2 : *Consider the query and document in Figure 8 again. After B_1 and C_1 are scanned, we maintain four lists,*

List("A_1",B)={B_1}, List("A_1",C)={C_1}, List("A_1A_2",B)={B_1}, List("A_1A_2",C)={C_1},

After B_2 and C_2 are scanned,

List("A_1",B)={B_1,B_2}, List("A_1",C)={C_1,C_2}, List("A_1A_2",B)={B_1,B_2}, List("A_1A_2",C)={C_1,C_2}.

After B_3 and C_3 are scanned, we delete "A_1A_2" from ComPath and output four binary tuples.

<“A_1A_2”, “B_1”>,<“A_1A_2”, “B_2”>,<“A_1A_2”, “C_1”>,<“A_1A_2”, “C_2”>.

Then List(“A_1”,B)={B_1,B_2,B_3}, List(“A_1”,C)={C_1,C_2,C_3}, List(“A_1A_3”,B)={B_3}, List(“A_1A_3”,C)={C_3}.

Subsequently, all lists are output.

Node B: <“A_1A_2”,“B_1”>,<“A_1A_2”,“B_2”>,<“A_1A_3”, “B_3”>, <“A_1”, “B_1”>,<“A_1”, “B_2”>,<“A_1”, “B_3”>

Node C: <“A_1A_2”,“C_1”>,<“A_1A_2”,“C_2”>,<“A_1A_3”,“C_3”>, <“A_1”, “C_1”>,<“A_1”, “C_2”>, <“A_1”, “C_3”>.

Note that the first elements of those binary tuples are sorted by the ascending post-order.

Finally, all binary tuples are merged and converted to the final solutions. □

Now we analyze the I/O complexity of our algorithm. The only operation we perform over lists is “append” (except for the final read out). We only need to access the tail of each list in memory as computation proceeds. Each list page is thus paged out only once, and paged back in again only when the list is ready for output. Therefore, the I/O cost required to maintain lists is proportional to the size of the output, provided that there is enough memory to hold the tail of each list in buffers.

5.4 Analysis of Algorithm BranchComPath

BranchComPath may produce a path solution that does not contribute to any final answer. This is because *candidate* strings might be inserted into *ComPath* and their fixed-end solutions might not be part of final solutions. Although candidate strings in *ComPath* cause the I/O sub-optimality, it is still necessary for us to insert candidate strings into *ComPath* in some cases, as illustrated below.

Consider the query β and the XML Doc2 in Figure 6(a),(c). The two streams S_A and S_D have A_1 and D_1 as their first element respectively, where MP($E_1B_1B_2A_1$, β) = {E_1B_1, $E_1B_1B_2$} and MP($E_1B_1B_2B_3D_1$, β) = {$E_1B_1B_2B_3$}. In this case, without advancing any stream, we cannot say if “E_1B_1”, “$E_1B_1B_2$” or “$E_1B_1B_2B_3$” is a common string in the MP sets of the labels from S_A and S_D, and we cannot advance any stream before knowing whether the strings in the MP sets of the current labels should be inserted into *ComPath*. So, in our approach, we insert “E_1B_1” and “$E_1B_1B_2$” into *ComPath* as candidate strings and output path solutions (E_1,B_1,A_1) and (E_1,B_2,A_1). Thus, we can safely advance stream S_A without missing any solutions. In the end, it is easy to verify that (E_1,B_2,A_1) contributes to the final answer, but (E_1,B_1,A_1) has not this property. Therefore, we insert candidate strings into *ComPath* to ensure the completeness of solutions in our algorithm.

However, if the *fan-out* (not all) edges of a branch pattern are *ancestor-descendant* relationships, then we can ensure that we only need to insert *qualified strings* into *ComPath*. For example, consider the query in the above example again. If we modify the fan-out edge BD into *ancestor-descendant* relationship, $MP(E_1B_1B_2B_3D_1,\beta)=\{E_1B_1, E_1B_1B_2, E_1B_1B_2B_3\}$. Thus, "$E_1B_1$" and "$E_1B_1B_2$" are inserted into *ComPath* as qualified (not candidate) strings. In this case, there is no candidate string in *ComPath*. Next, we conclude the observation by stating the following lemmas.

LEMMA 5.3 *Consider a branch pattern β, where all fan-out edges are ancestor-descendant relationship. Given two labels d_i and d_j in data streams S_{qi} and S_{qj}, respectively, if* $\forall a\in MP(d_i, \beta)\wedge\forall b\in MP(d_j, \beta)\wedge$ (a is a prefix of b)*, then* $a\in MP(d_j, \beta)$.

PROOF: *Since all fan-out edges are ancestor-descendant relationships in β, we assume that $L_{qj} = L_B//L_j$. Since $a \in MP(d_i)$, string a satisfy the pattern L_B. Since $b \in MP(d_j)$ and a is a prefix of b, asssume that $d_j = b \cdot e = a\cdot c\cdot e$, then string a is a prefix of d_j and e satisfy pattern L_j. Thus, $c\cdot e$ satisfy the pattern "$//L_j$". Therefore, according to the definition of matched-prefix set (see Definition 5.1), $a \in MP(d_j)$.* □

LEMMA 5.4 *Consider a branch pattern β, where all fan-out edges are ancestor-descendant relationships. In Algorithm BranchComPath, each string that is inserted into set* ComPath *is guaranteed to be a* qualified *string. In other words, in this case, line 4 of BranchComPath can be canceled.*

PROOF: *We prove the theorem by contradiction. Assume that there is a string p appended into ComPath as a candidate string in line 6. Then* $p\in MP(d_m,\beta)$. *By the definition of candidate strings, there is a stream S_{qi}, such that* $p\notin MP(getED(S_{qi}), \beta)$. *According to the condition in line 4,* $\exists s\in MP(getED(S_{qi}),\beta)$ *and* p *is a prefix of* s. *By the above Lemma 5.3,* $p\in MP(getED(S_{qi}),\beta)$*, which contradicts our assumption. This concludes the Proof.* □

Next, we shall discuss the correctness of Algorithm *BranchComPath*, and then we analyze its complexity.

LEMMA 5.5: *At every point of computation in Algorithm BranchComPath, for the current label d_i in stream S_{qi}, if* $\forall s\in MP(d_i,\beta)$ *such that* $s\notin$ComPath *and s is not a prefix of the largest string in the MP sets of the current labels, then a fixed-end solution of s to pattern L_B impossibly becomes part of any final solution.*

PROOF: *We prove this theorem by contradiction. Assume that there exists a string s such that s has fixed-end solutions which become part of final answers and* $s \notin ComPath$ *and s is not a prefix of the largest string in the MP sets of the current labels. Then s should be a common prefix of the labels from all data streams. In Algorithm BranchComPath, if s is the common string in the current MP sets, then in line 5, s will be appended into set ComPath as qualified string. Thus* $s \in ComPath$*, which contradicts the assumption. Otherwise, if* $s \notin ComPath$ *and s is not a prefix of the largest string (say string t which comes from the data stream* S_t*) in the MP sets of the current labels, then according to Property 5.1, whatever strings c and d are concatenated with s and t, it always holds that* $s \cdot c < t \cdot s$ *. Thus, it is impossible for s to be in the MP set of labels from data stream* S_t*. Thus, s is not part of final answers. Therefore, it also contradicts the assumption.*

By this lemma, set *ComPath* caches partial and total strings that present part of final solutions to pattern L_B. Moreover, it is easy to see that in line 7, for each label d_i in streams S_{qi}, if there exists a string s such that $s \in (MP(d_i) \cap ComPath)$, then the corresponding path solution of d_i to pattern L_{qi} will be output. These lead to the following correctness result:

THEOREM 5.1 *Given a branch pattern β and an XML database D, Algorithm BranchComPath correctly returns all answers for β on D.*

THEOREM 5.2 *Consider an XML database D and a query branch pattern β with only ancestor-descendant relationship in the fan-out edges. The worst case I/O complexity of BranchComPath is linear in the sum of the sizes of input and output lists. Further, the worst-case CPU complexity of BranchComPath is the sum of n*S and the size of the output list, where n is the length of the longest path in β and S is the sum of the lengths of all labels in the input lists. Finally, the worst-case space complexity of this algorithm is that the number of leaf nodes in β times the length of the longest path in D.*

6 Twig Join Algorithm

A straightforward way of computing answers to a query twig pattern is to (i)decompose the twig pattern into multiple branch patterns, (ii)use *branch join algorithm* to identify solutions to each individual branch, and then (iii) merge-join these solutions to compute the final answers to the query twig. Clearly, this approach has many intermediate results which may not be part of any final answer. So in the following, we seek to overcome this problem by using Algorithm *TwigComPath*.

Algorithm TwigComPath (T)

```
//Assume all elements have the same DocId. Let T be a twig pattern.
//phase 1
01  for q_i in leafnodes(T)
02    locateMatchingLabel(S_{q_i}, β_{q_i}); //assume q_i in β_{q_i}
03  β_act := β_1 ;  // β_1 is the first branch with postorder traversal
04  while (¬end(T)) {  //bottom-up matching each branch
05    if (β_act is not the top pattern)
06      {matchNonTop(β_act);
07      β_act := next(β_act); }//with postorder traversal order
08    else {matchTop(β_act); //on reaching top pattern
09      β_act := β_1 ; }   }//end while
//phase 2
10  MergeAllPathSolutions();
// D^β={the longest string in Temp^{β_i} ) | β_i∈subpatterns(β)} ∪ {getED(S_{q_i})|q_i∈leafnodes(β)}
```

Procedure matchNonTop(β)

```
01 advanceStreams (β);
02 if (∀d_i,d_j∈D^β, ∀x∈MP(d_i,β), ∀y∈MP(d_j,β):(x,y) is a prefix pair){
03  addQualified (β,Temp^β);
04  addCandidate (β,Temp^β);
05  copyFromTempToComPath(β);  }//end if
06 OutputSolutions (β);
```

Procedure matchTop(β)

```
01  advanceStreams(β);
02 if (∀d_i,d_j∈D^β, ∀x∈MP(d_i,β), ∀y∈MP(d_j,β):(x,y) is a prefix pair){
03   addQualified (β,ComPath^β);
04   addCandidate (β,ComPath^β);
05   insertStringsToSubComPath (β);
06   showBranchSolutions(β);  } //end if
07 else  OutputSolutions (β);
```

Procedure advanceStreams (β)

```
01  Let max be the maximal string in ⋃_{d_i∈D^β} MP(d_i, β) ;
02  for q_i in leafnodes(β)
03   if (∀s∈MP(getED(S_{q_i}),β):(s∈(Temp^β∪ComPath^β)∨s≠prefix(max))
04     locateMatchingLabel(S_{q_i}, β);
```

Procedure addQualified (β,S)

```
01  for p∈ ⋂_{d_i∈D^β} MP(d_i, β)  if (p∉S)  append p to set S;
```

Procedure addCandidate (β,S)

//Assume MP(d_m,β) contains the minimal string in $\bigcup_{d_i \in D^{\beta}} MP(d_i, \beta)$;

01 **for** p∈MP(d_m,β) **if** (p∉S) append p to set S;

Procedure copyFromTempToComPath(β)

01 **for** s in $Temp^{\beta}$

02 **if** ((∃p∈$ComPath^{parent(\beta)}$: p == prefix(s))∧ (s∉$ComPath^{\beta}$))

03 append s to $ComPath^{\beta}$;

Procedure insertStringsToSubComPath (β)

01 **for** β_i in subpattens(β) { copyFromTempToComPath(β_i);

02 insertStringsToSubComPath(β_i);} //recursive call

Procedure showBranchSolutions(β)

01 OutputSolutions (β);

02 **for** β_i in subpattens(β) showBranchSolutions(β_i); //recursive call

---

Figure 9 Algorithm TwigComPath

EXAMPLE 6.1 *Consider the query in Fig 10(a), which includes two branch patterns β_1 and β_2, and an XML document tree in Fig 9(b). If we directly used Algorithm BranchComPath to answer β_1 and β_2 sequentially, then the solution* <A_1,X_1,D_1,E_1,F_1> *to pattern β_1 would be the intermediate results that do not contribute to any final answer. However, in Algorithm TwigComPath, before* $R_1A_1X_1D_1$ *is inserted into* $ComPath^{\beta_1}$, *it should check whether* R_1A_1 *can be inserted to* $ComPath^{\beta_2}$. *Since* <R_1A_1, R_1A_2> *is not a prefix-pair,* R_1A_1 *cannot be inserted into* $ComPath^{\beta_2}$ *as a qualified or candidate string. Thus, TwigComPath avoids outputting the intermediate results and continues advancing streams to rematch pattern β_1. Subsequent steps insert "$R_1A_2Y_1D_2$", "R_1A_2" into* $ComPath^{\beta_1}$, $ComPath^{\beta_2}$ *respectively and output the corresponding path solutions* <A_2,Y_1,D_2,E_2><A_2,Y_1,D_2,F_2>,<A_2,B_1> *and* <A_2,C_1>. *Finally, all path solutions are merged to get the final result* <A_2,Y_1, D_2, E_2,F_2,B_1,C_1 >.

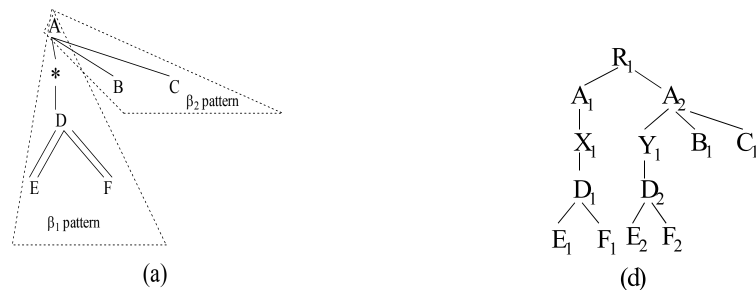


Figure 10 A sample query twig and a document tree

6.1 Analysis of Twig Join Algorithm

Next we show the correctness and complexity of this algorithm.

DEFINITION 6.1 (Depth) *Given a twig pattern T, if a branch pattern β of T is a leaf pattern, we define that the depth of β is 1. If β has some sub-patterns, say* $\beta_1', \beta_2', ... \beta_n'$, *then we define that the depth of β is max(* β_i' *) +1, where1 ≤i ≤n.*

LEMMA 6.1: *In Algorithm TwigComPath, given a twig pattern T, for any branch β of T, assume that the sequence of matched string which is returned from β to its parent pattern is {* $s_1, ... s_i ... s_j ... s_n$ *}. If* $s_j < s_i$ *but j>i , then* s_j *must be a prefix of* s_i.

PROOF [Sketch]: *We use an induction for the depth of β to prove it.*

Basic step: if depth(β)=1, then β is a leaf pattern. By using Property 5.1, it is easy to see the correctness of this lemma.

Inductive step: Assume that it holds as depth(β) ≤ i-1 , then we prove that it still holds for height(β) = i.

Since j>i, s_j *is returned later than* s_i. *So there exist the following three cases:*

Case (i): there exists at least a pair of labels t_i *and* t_j, *where* t_i *and* t_j *belong to the same leaf nodes or the same sub-pattern of pattern β, such that* s_i=prefix(t_i)∧s_j =prefix(t_j)∧(t_j>t_i). *This case is similar to the basic step.*

Case(ii): all labels t_i *and* t_j, *which belong to the same leaf nodes or the same sub-pattern of pattern β, are the same. It is impossible, for in such case,* $s_j = s_i$, *which is contradiction with the premise* $s_j < s_i$.

Case(iii): there exists at least a pair of labels t_i *and* t_j *for the same sub-pattern, such that* s_i=prefix(t_i)∧s_j =prefix(t_j)∧(t_j<t_i). *According to the assumption hypothesis,* t_j *is a prefix of* t_i *(for* t_j *and* t_i *are labels of the node in the pattern whose depth is i-1). So both* s_i *and* s_j *are prefixes of* t_i. *Moreover, by the premise* $s_j < s_i$, *we conclude that* s_j *is a prefix of* s_i. □

THEOREM 6.1. *Given a query twig pattern T and an XML database D. Algorithm TwigComPath correctly returns all answers for T on D.*

PROOF [Sketch]: *The soundness of this algorithm is similar to Theorem 5.1. Here we prove the completeness of this algorithm. We prove it by contradiction. Assume that these is a tuple* (t_1,..,t_n) *which satisfies the query pattern T, but* (t_1,..,t_n) *is not outputted. By the iterative property of Procedure MergeAllPathSolution(line 10), there must exist at least one branch pattern, say* β_j, *which misses a solution. Then there are two cases:*

(i) *There exists a string that is part of final solution to pattern* β_j *but is not inserted into* $ComPath^{\beta_j}$. *It is impossible, because, by Lemma 6.1, the sequence* $\{ s_1, \ldots s_i \ldots s_j \ldots s_n \}$ *returned from the child pattern to its parent pattern might not be sorted by lexicography order, but for each* $s_j < s_i$ *and* $j>i$, s_j *must be a prefix of* s_i. *If there is a prefix p of* s_j *can be inserted into* $ComPath^{\beta_j}$, *then p is guaranteed to be inserted when* s_j *is returned. Thus, p will not to be inserted to* $ComPath^{\beta_j}$ *in this case.*

(ii) *All strings that are part of final solution to pattern* β_j *are inserted into* $ComPath^{\beta_j}$, *but one of its corresponding path solutions is not output. In Procedure outputSolutions, for each label* d_i *in streams* S_{qi}, *if there exists a string s such that* $s \in (MP(d_i,\beta) \cap ComPath)$, *then the corresponding path solution of* d_i *to pattern* L_{qi} *will be output. So the second case is also impossible.*

THEOREM 6.2. *Consider an XML database D and a query twig pattern T with only ancestor-descendant relationship in the fan-out edges of each branch. The worst case I/O complexity of TwigComPath is linear in the sum of the sizes of input and output lists. Further, the worst-case CPU time complexity of TwigComPath is the sum of n*S and the size of the output list, where n is the length of the longest path in T and S is the sum of lengths of all labels in the input lists. Finally, the worst-case space complexity of this algorithm is that the number of leaf nodes in T times the length of the longest root-leaf path in D.*

It is important to note that the I/O cost of *TwigComPath* is usually much smaller than that of *TwigStack*[5], for *the former* only needs to scan the labels of query *leaf* nodes, while *the latter* needs to scan the labels of *all* nodes.

7. Using B+ tree

In this section, we propose the use of B^+ tree indices on the input lists to speed up the processing of query path, branch and twig patterns.

7.1 PathB+

We use B^+ tree indices to skip the elements that do not contribute to the final answers to speed up the processing of a query path pattern L. We assume that there exists a B+ index for the *leaf* node in pattern L, using (*DocId*, *ExDeweyID*) as the key. We also suppose that there is a non-leaf node in the query path, called a *reducer*, whose input list size is much smaller than that of the leaf

node. In Algorithm PathB$^+$(see Figure 11), we sequentially visit each label in the input list of the reducer . For each visited label s that matches the pattern from the *root* to the *reducer*(line 2) , we use B$^+$ index built on the leaf node to omit the examination of elements that are not final answers(in line 3), and then locate the elements with s as their prefix(line 4). When the input size of the reducer is significantly smaller than that of the leaf node, Algorithm *PathB*$^+$, which is evaluated in our experiment in Section 8, performs much better than that without using B$^+$ index.

Algorithm PathB$^+$ (L)

/* Let *q*, *r* denote the *leaf* node and the *reducer* of query path L. Let L_r denote the pattern from the *root* to the *reducer* of L. */

01 **while** $(\neg eof(S_r))$ {

02 locateMatchingLabel(S_r, L_r);

03 Use B+ tree in stream S_q to locate the smallest label that is larger than *getED*(S_r); // skip elements

04 **while** ((*getED*(S_r) == prefix(*getED*(S_q)) $\wedge$ $(\neg eof(S_q))$ {

05 **if** (*getED*(S_q) matches L) outputSolution(*getED*(S_q));

06 advance(S_q); } //end while

07 } //end while

Figure 11 Algorithm PathB$^+$

7.2 TwigComPathB+

To motivate the use of B$^+$ index in Algorithm *BranchComPath* and *TwigComPath*, consider the example in Fig 13(a),(b). If we used the previous *BranchComPath*, it would visit and analyze labels from E_2 to E_4 sequentially. Clearly, these steps are wasteful. We utilize B$^+$ index to skip E_2, E_3, E_4 as follows: after E_1 is scanned, directly go to E_5. Here E_5 is the node with the smallest label that has the common *root-branchpoint* (i.e. A_1/ B_1/ D_2) path with F_1.

Procedure advanceStreamsB$^+$(D)

01 Let *max* be the maximal string in $\bigcup_{d_i \in D^\beta} MP(d_i, \beta)$ by lexicography order;

(02) Let *min* be the shortest prefix of *max* in $\bigcup_{d_i \in D^\beta} MP(d_i, \beta)$;

03 **for** q_i in leafnodes(β)

04 **if** ($\forall$ p $\in$ MP(*getED*(S_{q_i}),β) : (p $\in$ ($Temp^\beta \cup ComPath^\beta$) $\vee$ p $\neq$ prefix(*max*))

(05) **if** (($ComPath^\beta \cap$ MP(*getED*(S_{q_i}),β)==∅)

(06) Use B+ index in stream S_{q_i} to locate the smallest label that is larger than *min*; //skip elements

07 **else** locateMatchingLabel(S_{q_i}, β);

Figure 12 Procedure advanceStreamsB$^+$

When we use B^+ tree in *BranchComPath* and *TwigComPath*, the main procedures remain unchanged. We only extend Procedure *advanceStream* to *advanceStreamB*$^+$ (see Fig 12. Here is the extension of *advancStream* in *TwigComPath*. The extension in *BranchComPath* is very similar). We assume that there exists a B^+ index in the input list of each leaf node, using (*DocId*, *ExDeweyID*) as the key.

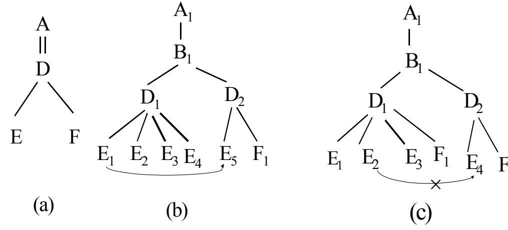


Figure 13 Using B+ tree to skip

The only changes appear in line 2, 5, 6 indicated by parentheses. In line 2, we define a variable *min* to be the shortest prefix of *max* in all MP sets. The variable *min* is to be used in line 6 to skip the elements that are not final answers. Line 5 gives a condition for correctly skipping elements. It requires no common string in *ComPath*$^\beta$ and MP(*getED*(S_{qi}),β). Otherwise, we might erroneously skip elements that may be part of answers. For example, in Figure 13(c), we scan E_1,F_1, and add $A_1B_1D_1$ to *ComPath*$^\beta$. Without the condition in line 5, the algorithm would directly jump from E_2 to E_4 and get the solution (A_1,D_2,E_4,F_2). And the solution (A_1,D_1,E_3,F_1) is missed. In line 6, we use B^+ index to skip elements that are smaller than *min*. The correctness of this step is guaranteed by the property on lexicography order mentioned in Property 5.1.

8. Experimental Results

In our experiments, we compared the query performance of *TwigComPath*, *TwigComPathB+*, *ViST* and *TwigStackXB*. We implemented all algorithms in C++. We used the B+ tree implementation in GiST[15] for all the indexes in *ViST* , *TwigComPathB+* and *TwigStackXB*.

8.1 Experimental Setting

All our experiments were performed on 1.7GHz Pentium 4 processor with 768MB RAM running on Linux system. The page size of 8K was used. For *ViST*, 8-byte number ranges were used to label the nodes in the virtual suffix tree. For *TwigStack* and *TwigStackXB*, 4-byte number ranges

were used to label nodes in the XML documents. As for *TwigComPath* and *TwigComPathB*$^{+}$, the variable length labels were used to implement the *extended Dewey ID* labeling scheme. Each label may have different length, consisting of a fixed-length (1 byte) stating the actual number of bytes of the label, and the label itself. Further, we used UTF-8 encoding[7] as an efficient way to present each label.

Table 1 Datasets

Dataset Name	Size (MB)	# of elements	# of attributes	# of values	Max/Avg depth
DBLP	134	3332130	404276	3576435	6/2.99
XMARK	118	1666315	381878	1593085	12/5.86

Two well-known datasets, namely DBLP[9] and XMARK[10], were used in our experiments, because they have different characteristics. DBLP is a real world dataset, where the document tree has good similarity in structure; while XMARK is a popular XML benchmark database, where the document tree has complicated structure. Table 1 shows additional information for DBLP and XMARK such as the maximum/average depth and the number of elements, etc.

The XPath queries listed in Table 2 were tested in our experiments. They have different characteristics in terms of selectivity, presence of values and twig structure. Table 2 also shows the number of twig (branch or path) occurrences for each query.

Table 2 XPath Queries

	Query	Dataset	# of Matches
Q1	*//inproceedings/author="Frank Manola"[year="1994"]*	DBLP	4
Q2	*//masterthesis/author*	DBLP	5
Q3	*//article[key="journals/spe/JankowitzKRS85"]/volume=15*	DBLP	1
Q4	*// inproceedings /title/sup[i]*	DBLP	58
Q5	*//parlist//text/emph="brain"*	XMARK	3
Q6	*//item/* [//bold="sold"] //text*	XMARK	11
Q7	*//item/mailbox[from]/mail[//bold/emph][//emph /keyword]*	XMARK	119
Q8	*//samerica/item[location]/mailbox/mail/text[keyword][bold]*	XMARK	509

8.2 Index Size

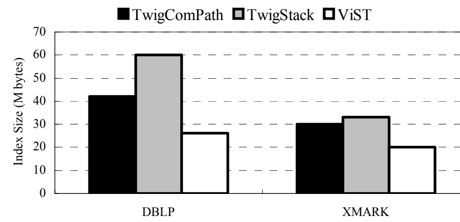


Figure 14 Index size

We compared the index sizes of three approaches, namely *TwigComPath*, *TwigStack* and *ViST*. As seen from Figure 14, the index size of *extended Dewey ID*(used in *TwigComPath*) is 5%-10% smaller than that of the *containment* labeling scheme(used in *TwigStack*). The size of ViST is the smallest among three approaches. This is because ViST use a novel indexing method to transform XML documents into the structure-encoded sequences. Note that although the *extended Dewey ID* has the worst-case space complexity of $O(N^2)$(considering a unary tree), N being the number of elements in a document tree, the real XML document is relatively balanced and does not have pathological structure to cause the labeling scheme to achieve the worst case bound.

We also compared the sizes of the input data between *TwigComPath* and *TwigStack*. As seen from Figure 15, the input data size of *TwigComPath* is usually significantly smaller than that of *TwigStack*. For example, in Q1,Q3,Q4,Q5,Q7, the size of data in *TwigComPath* is only about 10% of that in *TwigStack.* This can be easily explained as follows. In *TwigComPath*, we only need to access the labels of *leaf* nodes in a twig pattern, but in *TwigStack*, we need to visit the labels of *all* nodes.

The only exception is in query Q2, where both sizes are comparable. This is because the number of elements for the leaf node "author" of Q2 is much more than that of the non-leaf node "mastersthesis". Therefore, both algorithms have almost the same number of the input elements.

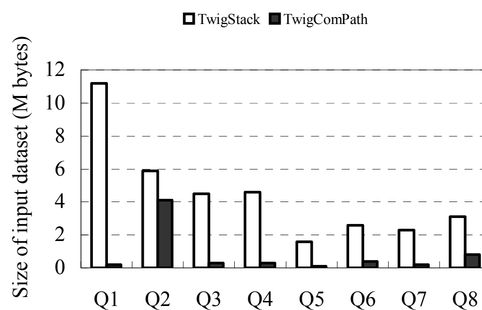


Figure 15 Input sizes of TwigStack Vs TwigComPath

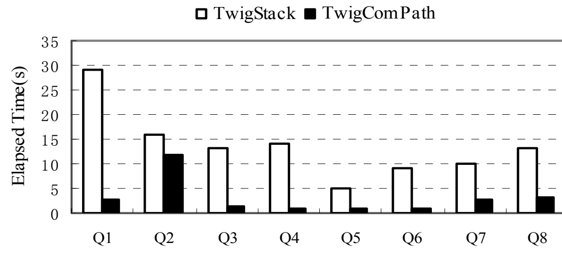


Figure 16 Performance result of TwigStack Vs TwigComPath

8.3 Performance Analysis

In Figure 16 and 17, we summarize the performance results for the queries listed in Table 2. We first discuss the benefits of *TwigComPath* over *TwigStack*.

8.3.1 *TwigComPath Vs TwigStack*

In this section, we shall compare the performance of *TwigComPath* and *TwigStack* in terms of the input data size, CPU overhead and the output data size.

Both *TwigComPath* and *TwigStack* need to sequentially scan the whole input dataset. As shown in Figure 16, the performance of *TwigComPath* is significantly better than that of *TwigStack* for most queries. The rationale is that the input data size of *TwigComPath* is much smaller than that of *TwigStack*. Thus, *TwigComPath* uses less time to scan the input data than *TwigStack* does.

As analyzed in Theorem 6.2, *TwigComPath* needs more CPU overhead than *TwigStack* does*,* because *TwigComPath* uses string matching algorithm for VFLDC. However, as shown in our experimental results, compared to the benefits from the great reduction of input data size, this overhead is usually insignificant.

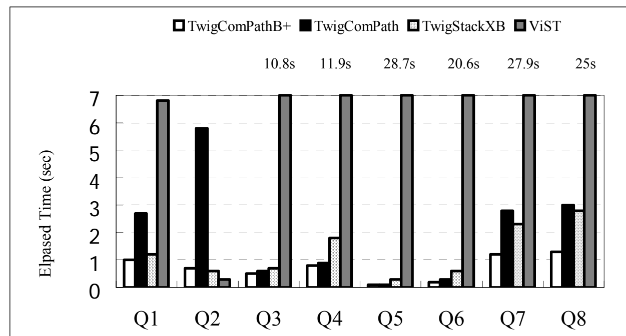


Figure 17 Performance result of TwigComPathB+, TwigComPath, TwigStackXB, ViST

Next we shall compare the output sizes between *TwigComPath* and *TwigStack.* Both algorithms have the similar output sizes for most queries. Because when there are only ancestor-descendant relationships in queries, both algorithms are I/O optimal. But if the queries (e.g. Q4,Q8) contain child-parent relationships in the fan-out edges, both algorithms become I/O sub-optimal. In such a case, *TwigComPath* usually outputs less intermediate results that do not contribute to the final answers than *TwigStack* does. For instance, in query Q4, consider a subtree in DBLP dataset, which consists of the element $title_1$ with the parent $inproceedings_1$ and two children i_1 and i_2, such that i_1 has a child sup_1. In the first phase of *TwigStack*, sup_1, i_1, i_2 are the descendants of $title_1$, so $(inproceedings_1, title_1, i_1)$ and $(inproceedings_1, title_1, i_2)$ are output as the path solutions. But they are pruned in the second phase, because there is no matching results in the subtree for Q4. However, in *TwigComPath*, these intermediate results are not output, for the matched-prefix set of element sup_1 is empty and we easily identify that there is no solution in the subtree.

Finally, note that *TwigComPath* can easily process queries with wildcards "*", but *TwigStack* is usually not efficient. For example, Q6 is a branch pattern with a wildcard "*". *TwigStack* cannot access the labels of only four nodes to answer the query, because it would cause false matching (please refer to Section 2). In our implementation, we use DTD constraints to know that only tags with *description* and *mailbox* can match "*". So *TwigStack* needs to visit the labels of at least six nodes (i.e. *description*, *mailbox, item*, *bold*, *sold, text)* to answer Q6, which results in more disk I/O's in *TwigStack*. On the contrary, the presence of wildcards does *not* add extra overhead in *TwigComPath* and we only need to access the labels of two nodes (*sold, text*) to answer the query.

8.3.2 *TwigComPath Vs TwigComPathB$^+$ and PathB$^+$*

In this section, we shall report the effects of using B$^+$ indices in our algorithms. Our results show that the total number of visited elements in algorithms that use B$^+$ indices is consistently smaller than those without using B$^+$ indices. For example, in Q1, there are 31923 elements tagged with "1994". *TwigComPath* must visit all these elements to find only 4 query answers. But in *TwigComPathB$^+$* algorithm, since there are only 43 elements tagged with "Frank Manola", a significant portion of "1994" elements is pruned. Thus, *TwigComPathB+* performs better than *TwigComPath* in this query. Similar is the case for query Q2, where the leaf node "author" occurs frequently (716488 times) in DBLP, but the non-leaf node "masterthesis" occurs occasionally (5 times). In *PathB$^+$*, we utilize "masterthesis" as a *reducer* to skip most of the elements tagged with "author". As we expected, the performance of *PathB$^+$* is much better than that of the algorithm without using B$^+$ tree in this query.

8.3.3 *TwigComPathB*$^{+}$ *Vs ViST*

For most queries, *TwigComPathB*$^{+}$ performed significantly better than ViST (see Figure 17). ViST used the *top-down* transformation of the query twig and data. The selectivity of leaf nodes in a query pattern is usually higher than that of the non-leaf nodes, which results in a large number of nodes in the virtual suffix tree being visited during subsequence matching for commonly occurring tags. For example, the non-leaf nodes "author", "year" in Q1 and "key", "volume" in Q3 suffer from this operation. In contrast, *TwigComPathB*$^{+}$ only needs to visit the labels of leaf nodes to perform string matching and comparing. The number of visited elements in *TwigComPathB*$^{+}$ is much smaller than that in ViST for most queries.

However, when the selectivity of non-leaf nodes is very high, ViST may perform efficiently. For instance, consider Q2, where there are very few occurrences of the tag "masterthesis" in DBLP and thereby very few of "author" exist in the virtual suffix tree. In such a query, ViST has a comparable performance with *TwigComPathB*$^{+}$.

On the other hand, it is also worth noting that the number of scanned elements is unnecessarily proportional to the processing performance. The rationale is that while few elements are distributed in different pages, each page should be read. But if many elements consecutively exist in the same page, we need to read only one page. For example, in Q3, the number of elements visited by *TwigComPathB*$^{+}$ is only 8, while Algorithm *TwigComPath* read all 2866 elements. However, the elapsed time of both algorithms is very similar.

8.3.4 *TwigComPathB+ Vs TwigStackXB*

In this section, we compare *TwigComPathB+* and *TwigStackXB*. Our experimental results show that, for most queries, they have comparable performances. But for queries Q6, Q7,Q8, *TwigComPathB+* is clearly better than *TwigStackXB*. We explain the reasons as follows.

When the join data scatter in the whole dataset, *TwigStackXB* usually needs to go deep in the XB-tree and thereby increases I/O cost. For example, in Q7, pattern *item/mailbox[from]/mail* frequently occurs in the XMARK. But a small portion of these patterns has two sub-patterns "*bold/emph*" and "*emph/keyword*" as the descendants of *mail*. In addition, the tags *bold* and *emph* occur frequently near *mail* in the dataset. As a result, *TwigStackXB* is forced to drill down to the lower regions (including leaves) of XB-tree several times in order to eliminate such partial answers. This process increases the disk I/O. However, *TwigComPathB+* is able to naturally avoid checking the partial matching, for it only visits the labels of two leaf nodes ("*emph"* and "*keyword"*) in the queries. And *TwigComPathB+* consistently omits the examination of elements that are not part of final answers independent of the distribution of join data. Finally, as mentioned before, *Twig-*

StackXB, like *TwigStack,* is also not efficient for the processing of queries with wildcards "*"(e.g. in Q6).

9. Conclusion and Future Work

In this paper, we have presented a new paradigm for XML pattern matching. We designed a novel Dewey ID labeling scheme which enable us to transform XML path pattern matching into string matching. We also developed Algorithm *BranchComPath* and *TwigComPath* to match branch and twig patterns respectively. Furthermore, the theoretical analysis and proof have been given to show the correctness and efficiency of our approach. Finally, we provided empirical data to demonstrate the performance benefit of our algorithms.

In fact, the efficiency of our algorithms mainly attributes to the *extended Dewey ID* labeling scheme. In the previous *containment*[19] labeling scheme, where each label consists of a 4-tuple (*DocId*,*LeftPos*:*RightPos,LevelNum*). From the label of one element, we only know its *DocId* and *LevelNum*. From the labels of two elements, we can identify their ancestor-descendant (and parent-child) relationship, but no more information is provided. However, with the *extended Dewey ID*, from the label of an element alone, we can use FST to derive the names of all elements in the path from root to this element. From the labels of two elements, we can immediately identify whether they match a branch pattern. As a result, we believe our study provides insight into the importance of designing a powerful labeling scheme for efficient XML query processing.

There are several avenues for future works. For instance, we can extend our algorithm to efficiently process the ordered twig patterns matching. We also plan to exploit DTD constrains to reduce the search space and further improve the performance of our twig pattern matching algorithm.

References

[1] Scott Boag, Don Chamberlin et al. Xquery 1.0: An XML Query Language W3C Working Draft 22 August 2003

[2] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources *In international Workshop WebDB*'2000 Dallas, TX, May page 53-62

[3] Anders Berglund , Scott Boag , et al. XML Path Language (XPath) 2.0 W3C Working Draft 22 August 2003

[4] Haixun Wang, Sanghyun Park, Wei Fan and Philip S. Yu ViST: A Dynamic Index Method for Querying XML Data by Tree Structures In *Proceedings of ACM SIGMOD* 2003 pages 110-121

[5] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching In *Proceedings of ACM SIGMOD* 2002 pages 310-321

[6] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras and Carlo Zaniolo Efficient Structural Joins on Indexed XML Documents In *Proc. 28th VLDB Conference* 2002 pages 263-274

[7] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, Storing and Querying Ordered XML Using a Relational Database System In *Proceedings of ACM SIGMOD* 2002 pages 204-215

[8] Raghav. Kaushik, Philip Bohannon, Jeffery F Naughton, and Henry.F. Korth. Covering index for Branching path queries In *Proceedings of ACM SIGMOD*, June 2002 pages 133-144

[9] University of Washington XML Repository. Available from http://www.cs.washington.edu/research/xmldatasets/.

[10] XMARK: XML-benchmark project. http://monetdb.cwi.nl/xml

[11] Q Li and B. Moon Indexing and querying XML data for regular path expressions In *Proc. 27th VLDB Conference* 2001 pages 361-370

[12] Paul. F. Dietz Maintaining order in a linked list. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing* pages 122-127 California May 1982

[13] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel. Structural Joins: A primitive for efficient XML query pattern matching, In *Proceedings of the* ICDE 2002 pages 141-152

[14] Dao Dinh Kha, Masatoshi Yoshikawa and Shunsuke Uemura A Structural Numbering Scheme for XML Data In *Proceedings of the* EDBT 2002 LNCS 2490 pages 91-108, 2002.

[15] J. Hellerstein, J. Naughton, and A. Pfeifer. Generalized search trees for database systems. In *Proceedings of the 21st VLDB*, pages 562-573, 1995.

[16] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proceedings of 27th VLDB* January 2001. pages 341-350

[17] Y. Papakonstantinou and V. Vianu DTD inference for views of XML data In *Proc. 19th ACM PODS* May 2000 pages 35-46

[18] T. Milo and D. Suciu. Index structures for path expressions In *the 7th International Conference on Database Theory (ICDT)* 1999 pages 277-295

[19] C. Zhang, J.F. Naughton, D.J. Dewitt, Q. Luo and G.M. Lohman, On Supporting containment Queries in Relational Database Management Systems In P*roceedings of. ACM SIGMOD*, 2001. pages 425-436

[20] R.Goldman and J. Widom Dataguides: Enabling query formulation and optimation in semistructured databases. In *Proceedings of the 23th VLDB* pages 436-445, 1997

[21] J.Min, C. Chung and K.Shim. APEX: An adaptive path index for XML data. In *Proceedings of the 2002 ACM-SIGMOD Conference*, Madison, Wisconsin, June 2002 pages 121-132

[22] M. Fernandez, D. Suciu Optimizing Regular Path Expressions Using Graph Schemas In *Proceedings of the 14th ICDE Conference*, Orlando, Florida, February 1998 pages 14-23

[23] G. Kucherov and M. Rusinowitch, Matching a Set of Strings with Variable Length Don't Cares, *Theoretical Computer Science* 178, 129-154, 1997 pages 129-15

Index of Definitions

DEFINITION 4.1 ***DTD,*** p7

DEFINITION 4.2 ***Function f(x,R(s)),*** p7

DEFINITION 4.3 ***Finite state transducer (FST)***, p10

DEFINITION 5.1 ***Matched-Prefix set,*** p12

DEFINITION 5.2 ***Fixed-end solution,*** p12

DEFINITION 5.3 ***Lexicography order,*** p13

DEFINITION 5.4 ***Post-order,*** p16

DEFINITION 6.1 ***Depth,*** p20