THE NATIONAL UNIVERSITY OF SINGAPORE

School of Computing

Kent Ridge Road, Singapore 119260

TR A6/05

From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching

Jiaheng LU, Tok Wang LING, Chee-Yong CHAN and Ting CHEN

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

June , 2005

TECHNICAL REPORT

Forward

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

From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching

Jiaheng Lu, Tok Wang Ling, Chee-Yong Chan and Ting Chen

National University of Singapore, Singapore

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

Abstract

Finding all the occurrences of a twig pattern in an XML database is a core operation for efficient evaluation of XML queries. A number of algorithms have been proposed to process a twig query based on *region encoding* labeling scheme. While region encoding supports efficient determination of ancestor-descendant (or parent-child) relationship between two elements, we observe that the information within a single label is very *limited*. In this paper, we propose a new labeling scheme, called *extended Dewey*. This is a *powerful* labeling scheme, since from the label of an element alone, we can derive all the elements names along the path from the root to the element. Based on *extended Dewey*, we design a novel holistic twig join algorithm, called TJFast. Unlike all previous algorithms based on region encoding, to answer a twig query, TJFast only needs to access the labels of the *leaf* query nodes. Through this, not only do we reduce disk access, but we also support the efficient evaluation of queries with wildcards in branching nodes, which is very difficult to be answered by algorithms based on region encoding. Finally, we report our experimental results to show that our algorithms are superior to previous approaches in terms of *the number of elements scanned*, *the size of intermediate results* and *query performance*.

1 Introduction

With the rapidly increasing popularity of XML for data representation, there is a lot of interest in query processing over data that conforms to a *tree-structured* data model. Since the data objects in a variety of languages(e.g. XPath, XQuery) are typically trees, twig (a small tree) pattern matching is the central issue.

In practice, XML data may be very large, complex and have deep nested elements. Thus, efficiently finding all twig patterns in an XML database is a major concern of XML query processing.

In the past few years, many algorithms ([2],[5],[9],[10]) have been proposed to match such twig patterns. These approaches (i) first develop a labeling scheme to capture the structural information of XML documents, and then (ii) perform twig pattern matching based on labels alone without traversing the original XML documents.

For solving the first sub-problem of designing a proper labeling scheme, the previous methods use a *tree-traversal* order(e.g. extended preorder [11]) or textual positions of *start* and *end* tags (e.g. region encoding [2]) or path expressions(e.g. Dewey ID [20]) or prime numbers (e.g. [23]). By applying these labeling schemes, one can determine the relationship (e.g. ancestor-descendant) between two elements in XML documents from their labels alone. Although existing labeling schemes preserve the positional information within the hierarchy of an XML document, we observe that the information contained by a single label is very *limited*. As an illustration, let us consider the most popular *region encoding*, where each label consists of a 3-tuple ($start$, end, $level$). Element a is an ancestor of element b if and only if $a.start<b.start$ and $a.end>b.end$. Given the labels of two elements, one can identify the ancestor-descendant, parent-child relationship and their document order, *but no more information is provided*.

In this paper, motivated by the existing *Dewey ID* [20], we propose a new *powerful* labeling scheme, called *extended Dewey ID* (for short, *extended Dewey*). The unique feature of this scheme is that, from the label of an element alone, we can *derive the names of all elements in the path from the root to this element*. For example, Figure 1 shows an XML document with *extended Dewey* labels. Given the label "0.5.1.1" of element *text* alone, we can derive that the path from the *root* to *text* is "*/bib/book/chapter/section/text*". An immediate benefit of this feature is that, to evaluate a twig pattern, we *only need to access the labels of elements that satisfy the* ***leaf*** *node predicates in the query*. Further, this feature enables us to easily match a path pattern by string matching. Take element "0.5.1.1" as an example again. Since we see that its path is "*/bib/book/chapter/section/text*", it is quite straightforward to determine whether this path matches a path query (e.g. "*//section/text*"). As a result, the *extended Dewey* labeling scheme provides us an *extraordinary* chance to develop a new efficient algorithm to match a twig pattern.

For solving the second sub-problem of performing structural joins efficiently, several algorithms have been developed to process twig queries. In particular, Bruno et al. [2] proposed the holistic twig matching algorithms PathStack/TwigStack. For evaluating queries with only *ancestor-*

descendant(A-D) edges, TwigStack guarantees that each intermediate path solution contributes to final answers. Lu et al.([12]) proposed TwigStackList to efficiently handle twig queries with *parent-child*(P-C) relationships.

Wildcard steps in XPath are commonly used when element names are unknown or do not matter([4]). Previous holistic twig matching algorithm is inefficient to answer queries with wildcards in branching nodes. For example, consider the XPath: //a/*[b]/c. By reading the region encoding of a, b and c, we cannot answer this query.[1] How can we answer such queries efficiently?

In this paper, we propose a novel holistic twig join algorithm, called TJFast(i.e. a Fast Twig Join algorithm) based on *extended Dewey* labeling scheme. To match a twig pattern, our algorithm only scans elements for query *leaf* nodes. This feature brings us two immediate benefits:(i) TJFast typically access much less elements than algorithms based on region encoding; and (ii) TJFast can efficiently process queries with wildcards in internal nodes. We make the following contributions:

- We propose to enhance *Dewey ID* labeling scheme by incorporating element-name (i.e. node-type) information. Our approach is based on using *modulo* function and a *finite state transducer*(FST) to derive the element names along a path.
- We develop a novel holistic twig join algorithm, called TJFast. When there are only A-D relationships between branching nodes and their children, TJFast is I/O optimal among all sequential algorithms that read the entire input. In other words, the optimality of TJFast allows the existence of P-C relationships between non-branching nodes and the children.
- We perform a comprehensive experiment to demonstrate the benefits of our algorithms over previous approaches.

Organization The rest of the paper proceeds as follows. We first discuss preliminaries in Section 2. The *extended Dewey* labeling scheme is presented in Section 3. We present TJFast algorithm in Section 4. Section 5 is dedicated to our experimental results and we close this paper by the related work and a conclusion.

[1]Note that even if b and c are descendants of a and their level difference with a is 2, b and c may not be query answers, as they do not have the common parent.

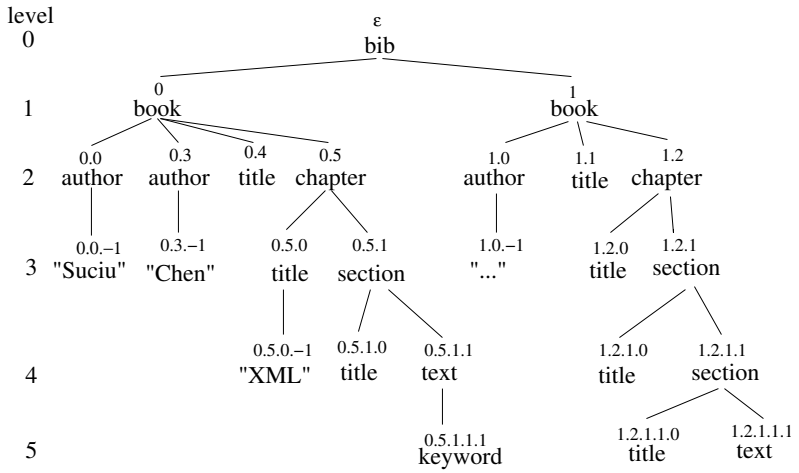


Figure 1: An XML tree with *extended Dewey* labels

2 Preliminaries

2.1 Data model and XML twig pattern

We model XML documents as *ordered* trees. Queries in XML query languages make use of twig patterns to match relevant portions of data in an XML database. The twig pattern node may be an element tag, a text value or a wildcard "*". The query twig pattern edges are either parent-child or ancestor-descendant edges. For convenience, we distinguish between query and data nodes by using the term "node" to refer to a query node and the term "element" to refer to a data element in a document.

Given a query twig pattern Q and an XML document D, a match of Q in D is identified by a mapping from the nodes in Q to the elements in D, such that: (i) the query node predicates are satisfied by the corresponding database elements, wherein wildcard "*" can match any single tag; and (ii) the parent-child and ancestor-descendant relationships between query nodes are satisfied by the corresponding database elements. The answer to query T with n nodes can be represented as a list of *n-ary* tuples, where each tuple $(t_1, \cdots, t_n)$ consists of the database elements that identify a distinct match of T in D.

2.2 Dewey ID labeling scheme

Tatarinov et al.[20] propose *Dewey ID* labeling scheme to present the position of an element occurrence in an XML document. In *Dewey ID*, each element is presented by a vector: (i) the root is labeled by a empty string ε; (ii) for a non-root element u, $label(u)= label(s).x$, where u is the x-th child of s. *Dewey ID* supports efficient evaluation of structural relationships between elements. That is, element u is an ancestor of element s if and only if $label(u)$ is a prefix of $label(s)$.

Dewey ID has a nice property: one can derive the ancestors of an element from its label alone. For example, suppose element u is labeled "1.2.3.4", then the parent of u is "1.2.3" and the grandparent is "1.2" and so on. With the knowledge of this property, we further consider that if the names of all ancestors of u can be derived from $label(u)$ alone, then XML path pattern matching can be directly reduced to string matching. For example, if we know that the label "1.2.3.4" presents the path "a/b/c/d", then it is quite straightforward to identify whether the element matches a path pattern (e.g. "//c/d"). Inspired by this observation, we develop an *extended* Dewey ID labeling scheme which provides an *extraordinary* chance for us to design a new algorithm to match XML path (and twig) pattern.

3 Extended Dewey and FST

In this section, we aim at extending *Dewey ID* labeling scheme to incorporate the element-name information. A straightforward way is to use some bits to present the element-name sequence with number presentation, followed by the *original Dewey* label. The advantage of this approach is simple and easy to implement. However, as shown in our experiments in Section 5, this method faces the problem of the large label size. In the following, we will propose a more concise scheme to solve this problem. In particular, we first *encode* the names of elements along a path into a single Dewey label. Then we present a *Finite State Transducer*(FST) to *decode* element names from this label. For simplicity, we focus the discussion on a single document. The labeling scheme can be easily extended to multiple documents by introducing document ID information.

```
<!ELEMENT bib (book*)>
<!ELEMENT book ( author+, title, chapter* ) >
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT chapter (title, section*)>
<!ELEMENT section (title, (text | section)*)>
<!ELEMENT text (#PCDATA | bold | keyword | emph) *>
<!ELEMENT bold (#PCDATA | bold | keyword | emph )*>
<!ELEMENT keyword (#PCDATA | bold | keyword | emph )*>
<!ELEMENT emph (#PCDATA | bold | keyword | emph) *>
```

Figure 2: DTD for XML document in Fig 1

3.1 Extended Dewey

The intuition of our method is to use *modulo* function to create a mapping from an integer to an element name, such that given a sequence of integers, we can convert it into the sequence of element names.

In the *extended Dewey*, we need to know a little additional schema information, which we call a *child names clue*. In particular, given any tag t in a document, the *child names clue* is all (distinct) names of children of t. This clue is easily derived from DTD, XML schema or other schema constraint. For example, consider the DTD in Figure 2; the tag of all children of *bib* is only *book* and the tags of all children of *book* are *author*, *title* and *chapter*. Note that even in the case when DTD and XML schema are unavailable, our method is still effective, but we need to scan the document once to get the necessary *child names clue* before labeling the XML document.

Let us use $CT(t) = \{t_0, t_1, \cdots, t_{n-1}\}$ to denote the *child names clue* of tag t. Suppose there is an ordering for tags in $CT(t)$, where the particular ordering is not important. For example,in Fig 3, $CT(book) = \{author, title, chapter\}$. Using child names clues, we may easily create a mapping from an integer to an element name. Suppose $CT(t) = \{t_0, t_1, \cdots, t_{n-1}\}$, for any element e_i with name t_i, we assign an integer x_i to e_i such that $x_i \ mod \ n = i$. Thus, according to the value of x_i, it is easy to derive its element name. For example, $CT(book) = \{author, title, chapter\}$. Suppose e_i is a child element of *book* and $x_i = 8$, then we see that the name of e_i is *chapter*, because $x_i \ mod \ 3 = 2$. In the following, we extend this intuition and describe the construction of *extended Dewey*

labels.

The *extended Dewey* label of each element can be efficiently generated by a *depth-first* traversal of the XML tree. Each *extended Dewey* label is presented as a vector of integers. We use $label(u)$ to denote the *extended Dewey* label of element u. For each u, $label(u)$ is defined as $label(s).x$, where s is the parent of u. The computation method of integer x in *extended Dewey* is a little more involved than that in the *original Dewey*. In particular, for any element u with parent s in an XML tree,

(1) if u is a text value , then $x = -1$;

(2) otherwise, assume that the element name of u is the k-*th* tag in $CT(t_s)$ (k=0,1,...,n-1), where t_s denotes the tag of element 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 of the left sibling of u is y (at this point, the left sibling of u has been labeled), then

$$x = \begin{cases} \lfloor \frac{y}{n} \rfloor \cdot n + k & \text{if } (y \bmod n) < k; \\ \\ \lceil \frac{y}{n} \rceil \cdot n + k & \text{otherwise.} \end{cases}$$

where n denotes the size of $CT(t_s)$.

EXAMPLE 3.1 *Figure 1 shows an XML document tree that conforms to the DTD in Figure 2. For instance, the label of chapter under book("0") is computed as follows. Here* $k = 2$ *(for chapter is the third tag in its child names clue, starting from 0),* $y = 4$ *(for the last component of "0.4" is 4), and* n=3*, so* $y \bmod 3 = 1 < k$*. Then* $x = \lfloor 4/3 \rfloor * 3 + 2 = 5$*. So chapter is assigned the label "0.5".* □

We show the space complexity of *extended Dewey* using the following theorem.

Theorem 3.1 *The extended Dewey does not alter the asymptotic space complexity of the original one.*

PROOF: According to the formula in (2.2),it is not hard to prove that given any element s, the gap between the last components of the labels for every two neighboring elements under s is no more than $|CT(t_s)|$. Hence, with the binary representation of integers, the length of each component i of

extended Dewey label is at most $log_2|CT(t_{s_i})|$ more than that of the *original Dewey*. Therefore, the length difference between an *extended Dewey* label with m components and an *original one* is at most $\sum_{i=1}^{m} log_2|CT(t_{s_i})|$. Since m and $|CT(t_{s_i})|$ are small, it is reasonable to consider this difference is a small constant. As a result, the *extended Dewey* does not alter asymptotic space complexity of the *original Dewey*.

3.2 Finite state transducer

Given the *extended Dewey* label of any element, we can use a *finite state transducer* (FST) to convert this label into the sequence of element names which reveals the *whole path from the root to this element*. We begin this section by presenting a function $F(t, x)$ which will be used to define FST.

DEFINITION 1. Let Z denotes the non-negative integer set and Σ denotes the alphabet of all distinct tag names in an XML document T. Given an tag t in T, suppose $CT(t) = \{t_0, t_1, \cdots, t_{n-1}\}$, a function $F(t, x)$: $\Sigma \times Z \rightarrow \Sigma$ can be defined by $F(t, x) = t_k$, where k= x *mod* n.

DEFINITION 2. (**Finite State Transducer**) Given *child names clues* and an *extended Dewey* label, we can use a deterministic *finite state transducer* (FST) to translate the label into a sequence of element names. FST is a 5-tuple (I, S, i, δ, o), where (i) the input set $I = Z \cup \{-1\}$; (ii) the set of states $S = \Sigma \cup \{PCDATA\}$, where $PCDATA$ is a state to denote text value of an element; (iii) the initial state i is the tag of the *root* in the document; (iv) the state transition function δ is defined as follows. For $\forall t \in \Sigma$, if $x = -1$, $\delta(t, x) = PCDATA$, otherwise $\delta(t, x) = F(t, x)$. No other transition is accepted. (v) the output value o is the current state name. □

EXAMPLE 3.2 *Figure 3 shows the FST for DTD in Fig 2. For clarity, we do not explicitly show the state for PCDATA here. An input of -1 from any state will transit to the terminating state PCDATA. This FST can convert any extended Dewey label to an element path. For instance, given an extended Dewey label "0.5.1.1", using the above FST, we derive that its path is "bib/book/chapter/section/text".* □

As a final remark, it is worth to note three points:(i) the memory size of the above FST is quadratic to the number of distinct element names in XML documents, as the number of transition

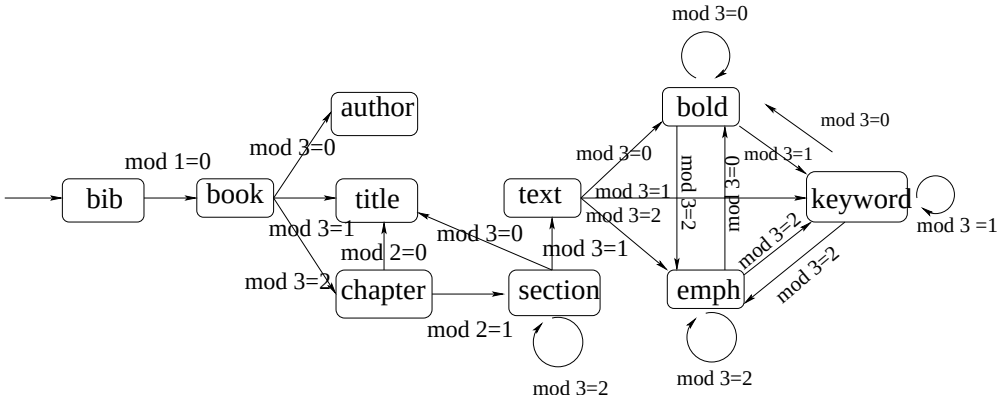


Figure 3: A sample FST for DTD in Fig 2

in FST is quadratic in the worst case; and (ii) we allow recursive element names in a document path, which is demonstrated as a loop in FST; and (iii) the time complexity of FST is linear in the length of an *extended Dewey* label, but independent of the complexity of schema definition.

3.3 Properties of extended Dewey

In this section, we summarize the following five properties of *extended Dewey* labeling scheme.

1. [**Ancestor Name Vision**] Given any *extended Dewey* label of an element, we can know all its ancestors' names (including the element itself).

2. [**Ancestor Label Vision**] Given any *extended Dewey* label of an element a, we can know all its ancestors' label.

3. [**Prefix relationship**] Two elements have *ancestor-descendant* relationships if and only if their *extended Dewey* labels have a prefix relationship.

4. [**Tight Prefix relationship**] Two elements a and b have *parent-child* relationships if and only if their *extended Dewey* labels $label(a)$,$label(b)$ have a *tight* prefix relationship. That is: (i) $label(a)$ is a prefix of $label(b)$; and (ii) $label(b).length\text{-}label(a).length$=1.

5. [**Order relationship**] Element a follows (or precedes) element b if and only if $label(a)$ is greater (or smaller) than $label(b)$ with lexicographeical order.

Region encoding also can be used for determining ancestor-descendant, parent-child and order relationships between two elements. But it cannot see the ancestors of an element and therefore

has not Properties 1 and 2. The original Dewey labeling scheme has Properties 2 to 5, but not Property 1. The first is unique for *extended Dewey*. Note that Property 1 and 2 are of paramount importance, since they provide us an *extraordinary* chance to efficiently process XML path (and twig) queries. For example, given a path query $a/b/c/d$, according to the **Ancestor Name and Label Vision Property**, we only need to read the labels of d to answer this query, which will significantly reduce I/O cost of previous algorithms based on region encoding. In the next section, we will use *extended Dewey* labels to design a novel and efficient holistic twig join algorithm, which utilizes the above five properties.

4 Twig Pattern Matching

4.1 Path matching algorithm

It is quite straightforward to evaluate a query path pattern in our approach. According to the Ancestor Name Vision property, we *only need to scan the elements whose tags appear in leaf node of query*. For each visited element, we first use FST to reveal the element names along the whole path, and then perform string matching against it. As a result, we evaluate the path pattern efficiently by scanning the input list once and ensure that each output solution is our desired final answer.

When path queries contain only parent-child relationships within the path, the string-matching can be processed very efficiently by simply comparing element names. When path queries contain ancestor-descendant relationships or wildcards "*" , the queries can be processed by string-matching with *don't care* symbols. There are a rich set of algorithms on efficient string processing with *don't care* symbols. (e.g. [16] and [8]).

It is worth noting that the I/O cost of our approach is typically much smaller than that of previous algorithms for path pattern matching (e.g. PathStack [2]), for we only scan labels for the query *leaf* node, while they need to scan elements for *all* query nodes.

4.2 Twig matching algorithm

This section presents a holistic twig pattern join algorithm, called TJFast. We will first introduce some data structures and notations.

4.2.1 Data Structures and Notations

Let q denote a twig pattern and p_n denote a path pattern from the *root* to the node $n \in q$. In our algorithms, we make use of the following query node operations: isleaf: Node → Bool; isBranching: Node → Bool; leafNodes: Node → {Node}; directBranchingOrLeafNodes: Node → {Node}. leafNodes(n) returns the set of leaf nodes in the twig rooted with n. directBranchingOrLeafNodes(n)(for short, dbl(n)) returns the set of all branching nodes b and leaf nodes f in the twig rooted with n such that in the path from n to b or f(excluding n,b or f) there is no branching nodes. For example, in the query Q1 of Fig 4, dbl(a)={b,c} and dbl(c)={f,g}.

Associated with each leaf node f in a query twig pattern there is a stream T_f. The stream contains *extended Dewey* labels of elements that match the node type f. The elements in the stream are sorted by the ascending lexicographical order. For example, "1.2" precedes "1.3" and "1.3" precedes "1.3.1". The operations over a stream T_f include $current(T_f)$, $advance(T_f)$ and $eof(T_f)$. The function $current(T_f)$ returns the *extended Dewey* label of the current element in the stream T_f. The function $advance(T_f)$ updates the current element of the stream T_f to be its next element. The function $eof(T_f)$ tests whether we are in the end of the stream T_f. We make use of two self-explanatory operations over elements in the document: $ancestors(e)$ and $descendants(e)$, which return the ancestors and descendants of e, respectively (both including e).

Algorithm TJFast keeps a data structure during execution: a set S_b for each branching node b. Each two elements in set S_b have an *ancestor-descendant* or *parent-child* relationship. So the maximal size of S_b is *no more than the length of the longest path* in the document. Each element cached in sets likely participates in query answers. Set S_b is initially empty.

4.2.2 TJFast

Algorithm TJFast, which computes answers to a query twig pattern q, is presented in Algorithm 1. TJFast operates in two phases. In the first phase (line 1-9), some solutions to individual root-leaf path patterns are computed. In the second phase (line 10), these solutions are merge-joined to compute the answers to the query twig pattern.

Given the *extended Dewey* label of an element, according to the **Ancestor Name Vision** property, it is easy to check whether its path matches the individual root-leaf path pattern. Thus,

Algorithm 1 TJFast

1: **for each** $f \in$ leafNodes(root)
2: locateMatchedLabel(f)
3: **endfor**
4: **while** ($\neg end(root)$) **do**
5: f_{act} = getNext($topBranchingNode$)
6: outputSolutions(f_{act})
7: advance($T_{f_{act}}$)
8: locateMatchedLabel(f_{act})
9: **end while**
10: mergeAllPathSolutions()

Procedure locateMatchedLabel(f)

/* Assume that the path from the root to element $get(T_f)$ is $n_1/n_2/\cdots/n_k$ and p_f denotes the path pattern from the *root* to *leaf* node f */

1: **while** $\neg((n_1/n_2/\cdots/n_k$ matches pattern $p_f) \wedge (n_k$ matches $f))$ **do**
2: advance(T_f)
3: **end while**

Function $end(n)$

1: Return $\forall f \in leafNodes(n) \rightarrow eof(T_f)$

Procedure outputSolutions(f)

1: Output path solutions of $current(T_f)$ to pattern p_f such that in each solution s, $\forall e \in s$:(element e matches a branching node $b \rightarrow e \in S_b$)

Algorithm 2 getNext(n)

1: **if** (isLeaf(n)) **then**
2: return n
3: **else**
4: **for each** $n_i \in$ dbl(n) **do**
5: $f_i = getNext(n_i)$
6: **if** ($isBranching(n_i) \bigwedge empty(S_{n_i})$)
7: return f_i
8: $e_i = max\{p | p \in MB(n_i, n)\}$
9: **end for**
10: max = $maxarg_i\{e_i\}$
11: min = $minarg_i\{e_i\}$
12: **for each** $n_i \in$ dbl(n) **do**
13: **if** ($\forall e \in MB(n_i, n) : e \notin ancestors(e_{max})$)
14: return f_i;
15: **endif**
16: **end for**
17: **for each** $e \in$ MB(n_{min}, n)
18: **if** ($e \in ancestors(e_{max})$) updateSet(S_n, e)
19: **end for**
20: return f_{min}
21: **end if**

Function MB(n, b)

1: **if** (isBranching(n)) **then**
2: Let e be the maximal element in set S_n
3: **else**
4: Let $e =$ current(T_n)
5: **end if**
6: Return a set of element a that is an ancestor of e such that a can match node b in the path solution of e to path pattern p_n

Procedure clearSet(S, e)

1: Delete any element $a \in S$ that is not any ancestor or descendant of e

Procedure updateSet(S, e)

1: clearSet(S,e)
2: Add e to set S

the key issue of TJFast is to determine whether a path solution can contribute to the solutions for the whole twig. In the optimal case, we only output the path solution that is merge-joinable to at least one solution of other root-leaf paths. Intuitively, if two path solutions can be merged, the necessary condition is that they have the common element to match the *branching* query node. For example, consider a simple query a[./b]/c and two path solution (a_1, b_1) and (a_2, c_1). Observe that two solutions can be merged only if $a_1 = a_2$. Therefore, in TJFast, in order to determine whether a path solution contributes to final answers, we try to find the most likely elements that match branching nodes b and store them in the corresponding set S_b.

It is not difficult to understand the main procedure of TJFast(see Algorithm 1). In line 1-3, for each stream, we use Procedure locateMatchedLabel to locate the first element whose path matches the individual root-leaf path pattern. In line 5, we identify the next stream $T_{f_{act}}$ to be processed by using *getNext(topBranchingNode)* algorithm, where *topBranchingNode* is defined as the branching node that is an ancestor of all other branching nodes(if any). In line 6, we output some path matching solutions in which each element that match any branching node b can be found in the corresponding set S_b. We advance $T_{f_{act}}$ in line 7 and locate the next matching element in line 8.[2]

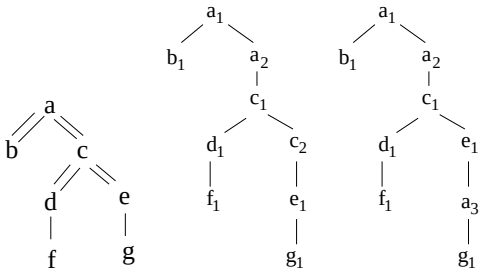
Algorithm *getNext*(see Algorithm 2) is the core function called in TJFast, in which we accomplish two tasks. The first is to identify the next stream to process; and the second is to update the sets S_b associated with branching nodes b, discussed as follows.

For the first task to identify the next processed stream, Algorithm *getNext(n)* returns a query leaf node f according to the following recursive criteria (i) if n is a leaf node, return n(line 2); else (ii) n is a branching node, then for each node $n_i \in dbl(n)$, (1) if the current elements cannot form a match for the subtree rooted with n_i, we immediately return f_i(line 7); (2) if the current element from stream T_{f_i} does not participate in the solution involving in the future elements in other streams, we return f_i(line 14); (3) otherwise we return f_{min} such that the current element e_{min} has the minimal label in all e_i by lexicographical order(line 20).

For the second task, we update set e_b. This operation is important, since the elements in e_b decides which path solution can be output in Procedure *outputSolutions*. In line 18 of Algorithm 2,

[2]Note that the second condition "n_k matches f" in line 1 of locateMatchedLabel is necessary, which avoids outputting duplicate solutions. For example, consider the element e with the path "$a_1/b_1/c_1/b_2$" and the path query "a/b". "$a_1/b_1/c_1/b_2$" can matches "a/b", but this solution has been output by another element ends with b_1.

before an element e_b is inserted to the set S_b, we ensure that e_b is an ancestor of (or equals) each other element e_{b_i} to match node b in the corresponding path solutions.



(a)Q1 (b) Doc1 (c) Doc2

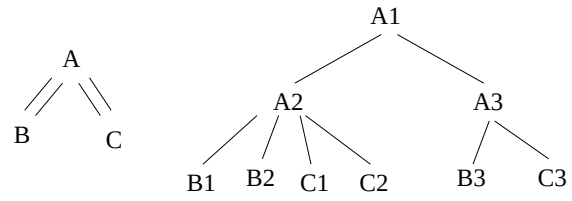
Figure 4: Example twig query and documents

EXAMPLE 4.1 *Consider Q1 and Doc1 in Fig 4(a-b). A subscript is added to each element in the order of pre-order traversal for easy reference. There are three input streams T_b, T_f and T_g. Initially, $getNext(a)$ recursively calls $getNext(b)$ and $getNext(c)$ (for $b, c \in dbl(a)$ in Q1). Since b is a leaf node in Q1, $getNext(b)$=b. Observe that MB(f,c)=$\{c_1\}$ and MB(g,c)=$\{c_1,c_2\}$, So $e_{max} = g$ and $e_{min} = f$ in line 10 and 11 of Algorithm 2. In line 18, c_1 is inserted to set S_c. Then, $getNext(c)$=f. Subsequently, a_1 is inserted to S_a and $getNext(a)$=b. Finally path solutions (a_1, b_1),(a_1, c_1, d_1, f_1) and (a_1, c_1, e_1, g_1) are output and merged. Note that although (a_1, c_2, e_1, g_1) matches the individual path pattern $a//c//e/g$, it is not output for $c_2 \notin S_c$.* □

Note that the second phase(line 10 of Algorithm 1) of TJFast can be performed efficiently, only when the intermediate path solutions are output in sorted order. To achieve this purpose, we would need to "block" some answers. The details of how to achieve this naturally in the scenario of TJFast are described in the next section.

4.3 Blocking techniques

Consider the simple query and dataset in Fig 5 (a) and (b). When Algorithm TJFast scan B_1 and C_1 and insert A_1, A_2 to set S_A , we cannot immediately output solutions $< A_2, B_1 >$ and $< A_2, C_1 >$. This is because there remains the possibility of a new element after B_1 or C_1 which joins with A_1 as long as A_1 is in set S_A. Therefore, we cannot output $< A_2, B_1 >$ and $< A_2, C_1 >$ until A_1 is deleted from the set. We now propose a procedure to guarantee the output path solutions



(a) Query (b) Document

Figure 5: An example of XML data that needs blocking

(a) $\{C_1, C_2, ..C_n\}$ C is a branhcing Node	(b) $\{P_1, P_2, ..P_m\}$ $\{C_1\}$ Both C and P are branching nodes and P is the direct ancestor node of C in the query.	(c) $\{C_1\}$ C is the top branching node
$(C_1.S + C_1.I)$ to $C_2.I$ where C_2 is the nearest ancestor node of C_1 in this set	$(C_1.S + C_1.I)$ to each $P_i.S$ where P_i is an ancestor of C_1	Output $(C_1.S + C_1.I)$

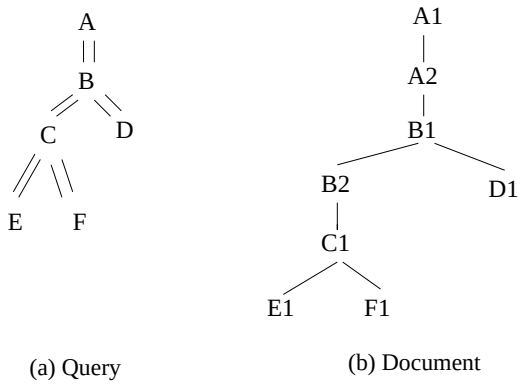
Figure 6: Possible set configuration when blocking results

are sorted, which is partly inspired by [2].

For this purpose, we maintain two lists associated with each element n in sets: the first, (S)elf-list, represents all blocked solution with root element n , and the second (I)nherit- list, represents all blocked solutions with root elements that are descendants of n. When an element n is inserted to a set, for each stream T_q , we initialize a list for each n and q. At any point of the algorithm, we do not directly output path solutions for any element, but add it to the Self-list of its responding nearest branching node. For example, in Fig 5(a) and (b), we scan B_1. Then add $< A_1, B_1 >$ to the Self-list of element A_1 and $< A_2, B_1 >$ to the Self-List of A_2.

In particular, suppose we are deleting element C_1 from the set. Depending on the current configuration, we proceed as follows(see Fig 6):

- (a) Element C_1 is not the only element in set, but has an ancestor C_2 . In this case, we first identify C_2, which is the nearest ancestor of C_1. Then we append the Self-list and Inherit-list of C_1 to the Inherit-list of element C_2.
- (b) Node P is the nearest ancestor of node C. Element C_1 is the only element in the set. In



(a) Query (b) Document

Figure 7: Illustration to blocking

this case, we append the Self-list and Inherit-list of C_1 to the Self-list of each element P_i , where P_i is an ancestor of C_1.

- (c) Node C has no ancestor that is a branching node in query. Element C_1 is the only element in the set. In this case, we output the contents of the self-list and inherit-list of element C_1.

Note that before the second phase of merge-join, unlike [2], our path solutions only involve in elements that match branching nodes. After the path solutions are merged, we can easily extend them to the full query solutions. This can be achieved because of the unique feature of extended Dewey label. The benefit of this approach is to reduce the size of intermediate results. We use the following two examples to illustrate the blocking techniques described above.

Example 4.2 Consider the query and data set in Fig 5 again. Initially, B_1 and C_1 are scanned. We do not immediately output their path solutions, but add them to the respective Self-lists. Subsequently, the path solutions of B_2,C_2 are also added to Self-lists. Then after B_3 and C_3 are scanned, we delete A_2 from its set. At this point, according to the rules in Fig 6(b), all elements in the Self-list of A_2 (here the Inherit-list of A_2 is empty) are appended to the Inherit-list of A_1. Finally, when A_1 is output from the set, all path solutions in the Self- and Inherit-lists of A_1 are sorted.

Example 4.3 Consider the query and data set in Fig 7. This example is used to illustrate that before the final merge, the path solutions only include elements that match branching nodes and leaf nodes. With the blocking technique of TJFast, we output three path solutions: $< B_1, C_1, E_1 >$, $< B_1, C_1, F_1 >$ and $< B_1, D_1 >$. Note that there is no node to match A. After we merge three solutions

to one solution $< B_1, C_1, E_1, F_1, D_1 >$, we extend it to two final solutions $< A_1, B_1, C_1, E_1, F_1, D_1 >$, $< A_2, B_1, C_1, E_1, F_1, D_1 >$. This can be achieved because we can derive the existence of A_1 and A_2 from the extended Dewey label of B_1.

Now we analyze the I/O complexity of our method. The only operation we perform over lists is "append" (except 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.

4.4 Analysis of TJFast

Next, we first show the correctness of TJFast and then analyze its complexity.

Lemma 1. *In Procedure clearSet of Algorithm TJFast, any element* e *that is deleted from set* S_b *does not participate in any new solution.*

PROOF: Suppose that on the contrary, there is a new solution using element e. Since e has not ancestor-descendant relationship with the new inserted element e_{new}, according to the Order Property, $label(e) < label(e_{new})$ by lexicographical order. Note that if $a<b$ and a is not a prefix of b, then whatever postfix c, d is attached to a and b respectively, $a.c<b.d$ holds. Therefore, $label(e)$ will not be a prefix of subsequent elements in any stream, which contradicts that e participates in a new solution. □

Lemma 2. *In line 18 of Function getNext, if element* e $\notin$ *ancestors*(e_{max}) *and* e $\notin S_n$ *, then* e *is guaranteed to not involve in any final solution.*

PROOF (**Induction on the number of calls to *getNext***): Consider the first call to $getNext$ for branching node n. Observe that set S_n is empty before this call. Since element e is not a prefix of e_{max}, e cannot become a prefix of any element in stream $T_{f_{max}}$. Therefore e does not participate in any final solution. For subsequent calls to *getNext*, we proceed as follows. Since element e is not a prefix of e_{max}, e cannot involve in the solutions of the future elements in stream $T_{f_{max}}$. So the only possible case is that e participates in the solution for the previous elements. But now e does

not appear in set S_b. Then either e is never added into set S_b or it has been wrongly deleted from set S_b. In the first case, according to the inductive hypothesis, element e does not participate in any final solution. The second case is impossible, since by Lemma 1, each deletion operation is *safe*. Therefore, the lemma is proved. □

Lemma 1 shows that any element deleted from sets does not participate in new solutions, so the deletion is *safe*. Lemma 2 shows that for any element e that matches a branching node, if e participates in any final answer, then e occurs in the corresponding set. Thus the insertion is *complete*. The two lemmas are important to establish the correctness of the following theorem.

Theorem 1. *Given a twig query q and an XML database D, Algorithm TJFast correctly returns all the answers for q on D.*

While the correctness holds for any given query, the I/O optimality holds only for the case where there are only *ancestor-descendant* relationships between *branching* nodes and their children.

Theorem 2. *Consider an XML database D and a twig query q with only ancestor-descendant relationships between branching nodes and their children. The worst case I/O complexity of TJFast is linear in the sum of the sizes of input and output lists. The worst-case space complexity is $O(d^2 * |b| + d * |f|)$, where $|f|$ is the number of leaf nodes in q, $|b|$ is the number of branching nodes in q and d is the length of the longest label in the input lists.*

PROOF: We first prove the I/O optimality. The following observation is important to prove the optimality of TJFast: if all branching edges are only *ancestor-descendant* relationships, then in line 18 of *getNext*, since $e \in \text{ancestors}(e_{max})$, $e \in \text{MB}(n_i, n)$ for each $n_i \in dbl(n)$. That is, e is guaranteed to be a **common** element in each current path solution. Note that we only output path solutions, in which elements that match branching nodes occur in the corresponding set(line 6 of Algorithm 1). Therefore, each intermediate path solution output in TJFast is guaranteed to contribute to final results when the query contains only *ancestor-descendant* relationships in *branching* edges.

As for space complexity, our result is based on the observation that in the worst case, the number of elements in branching node set S_b is at most d, where d is the length of the longest label in the input lists. Considering each extended Dewey label repeats its prefix, the total space complexity of S_b is $\text{O}(d^2)$. □

Theorem 2 holds only for query with *ancestor-descendant* relationships to connect *branching* nodes. Unfortunately, in the case where the query contains *parent-child* relationships between *branching* nodes and their children, Algorithm TJFast is no longer guaranteed to be I/O optimal. For example, consider a query $a[./b]/c$ and a data tree consisting of a_1, with children(in order) b_1, a_2, c_2, such that a_2 has children b_2, c_1. There are two streams T_b, T_c in TJFast and their first elements are b_1 and c_1 respectively. In this case, b_1 and c_1 are "locked" simultaneously, because we cannot advance any stream before knowing if it participates in a solution. Thus, optimality can no longer be guaranteed.

4.5 Comparison among TJFast, TwigStack and TwigStackList

In this section, we use the following example to illustrate the advantages of TJFast over TwigStack and TwigStackList.

EXAMPLE 4.2 Consider the query and data tree *Doc2* in Fig 4(a) and (c). There are three input streams T_b,T_f and T_g in TJFast. Initially, the current elements are b_1,f_1 and g_1. TJFast does not insert c_1 to set S_c, since by reading the label of g_1 alone, we immediately identify that g_1 does not contribute to query answers(for $a_1/a_2/c_1/e_1/a_3/g_1$ does not match $a//c//e/g$). In contrast, TwigStack pushes c_1 to *stack* S_c and outputs two "*useless*" intermediate path solution $<a_1, b_1>$ and $<a_1, c_1, d_1, f_1>$. The behavior of TwigStack is also reasonable because based on *region coding* of g_1, one cannot decide whether g_1 has the parent tagged with e. But based on *extended Dewey*, one can easily identify that the parent of g_1 is tagged with a rather than e. This example shows the benefit of *extended Dewey* labeling scheme on efficient processing of XML twig pattern matching.

Compared to TwigStack, TwigStackList looks more "clever". In the above example, TwigStackList does not hastily push c_1 to stack , but first checks the parent-child relationship between e_1 and g_1. Then they find that e_1 is not the parent of g_1. Then TwigStackList caches e_1 in a list and reads more elements in T_e. In this simple case, e_1 is the only element in stream T_e. So unlike TwigStack, TwigStackList does not output any useless intermediate results. Compared to TJFast, TwigStackList is also I/O optimal in this example, but TwigStackList needs to read more elements from all non-leaf node streams and its processing will be very complicated when g_1 has more than one ancestor tagged with e. (More examples about TwigStackList can be found in [12]) □

5 Experimental evaluation

5.1 Experimental setup

5.1.1 Testbed and Data set

We implemented four XML twig join algorithms: TJFast, TwigStack, TwigStackList and iTwigJoin in JDK 1.4 using the file system as a simple storage engine. Only TJFast is based on *extended Dewey* labeling scheme, and the other three use *region encoding*.

The reason that we choose these three algorithms for comparisons is that TwigStack, TwigStackList and iTwigJoin are efficient for different applications. TwigStack[2] is very efficient when query contains only ancestor-descendant relationships. TwigStackList[12] is efficient on answering queries with parent-child relationships. Finally, unlike the above two algorithms, which partition elements to one stream according to their tags alone, iTwigJoin[5] is a general twig join algorithm, which can be used on different data partition approaches. [5] researched two new data partitions: *tag+level* and *prefix path streaming* (PPS). Such *refined* data partition strategies enable iTwigJoin to reduce I/O cost by pruning irrelevant data streams.

All experiments were run on a 1.7G Pentium IV processor with 768MB of main memory and 2GB quota of disk space, running windows XP system. We use four different datasets, including two synthetic and two real datasets. The first synthetic data is the well-known XMark benchmark data (with factor 5). The second is a random data set. We used ten different labels, namely A_1,A_2,...,A_{10}. The node labels in the tree were uniformly distributed. The two real datasets are DBLP and TreeBank[14][3]. We choose these two datasets since they have different characteristics. DBLP is a shallow and wide document, but TreeBank has very deep recursive structure. Table 1 summarizes their characteristics.

5.1.2 UTF-8 encoding

In our experiments, *extended Dewey* labels are not stored by the dotted-decimal strings displayed (e.g. "1.2.3.4"), but rather a compressed binary representation. In particular, we used UTF-8

[3]Since there is no DTD available for TreeBank and random data, we first scan this document once to get the *child names clue* of each tag.

Table 1: XML Data Sets (XM: XMark,TB:TreeBank)

	XM	Random	DBLP	TB
Data size(MB)	582	90	130	82
Nodes(million)	8	5.1	3.3	2.4
Max/Avg depth	12/5	10/5.1	6/2.9	36/7.8

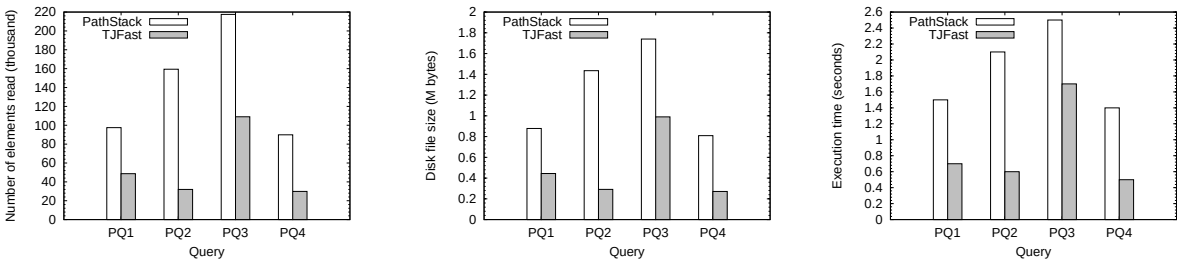
Table 2: Labels size (XM: XMark,TB:TreeBank)

	XM	Random	DBLP	TB
Original Dewey(MB)	56.2	36.1	18.1	22.8
Region coding(MB)	71.9	45.2	21.6	23.3
Naive extension(MB)	92.9	55.8	27.7	41.9
Extended Dewey(MB)	72.6	43.3	19.5	28.7

encoding as an efficient way to present the integer value, which was proposed by Tatarinov et al. [20]. Our experimental results show that compared to the naive implementation, where each integer value is presented as a fixed number of bytes, the UTF-8 encoding can save about 50% space cost.

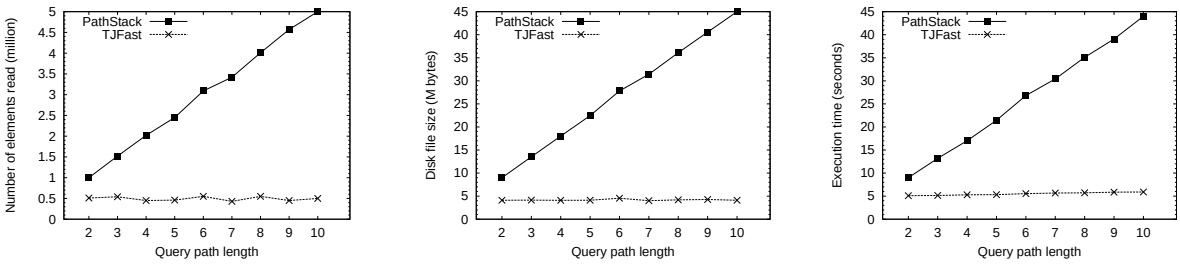
5.1.3 Labels size

We compare the labels size of four labeling schemes in Table 2. Our first conclusion is that the size of the *naive extension*, which directly presents the element-name sequence in number presentation ahead of the *original Dewey* labels, is generally larger than that of our *extended Dewey* labeling scheme. Our second conclusion is that when the document tree is shallow and wide (i.e. DBLP), the size of *extended Dewey* is smaller than that of *region encoding*. But while the document tree is deep(i.e. TreeBank), the size of *region encoding* is smaller. This is because *extended Dewey* is a variation of prefix labeling scheme, whose size is closely related to the average depth of documents. Our third conclusion is that the size of *extended Dewey* is about 10%-30% more than that of *original Dewey*. As we will show in our experiments, it is worth using this additional space-overhead, since it significantly improves the performance of XML twig pattern matching.



(a)Number of elements scanned (b) Size of disk files scanned (c) Execution time

Figure 8: PathStack versus TJFast using XMark data



(a)Number of elements scanned (b) Size of disk files scanned (c) Execution time

Figure 9: PathStack versus TJFast using random data

5.2 Performance Analysis

5.2.1 Path Queries

In the first experiment, we compare our algorithm TJFast with the previous work PathStack to match path pattern without branching nodes. For this purpose we first use XMark benchmark data and four path queries[4] shown in Table 3. Figure 8(c) shows the execution time for two algorithms. We also show the number of elements scanned and the size of disk files read by two algorithms in Figure 8(a)(b).

An immediate observation from the figures is that TJFast is more efficient than PathStack. In particular, PathStack could perform 400% more disk I/Os than those required by TJFast (e.g. PQ2).

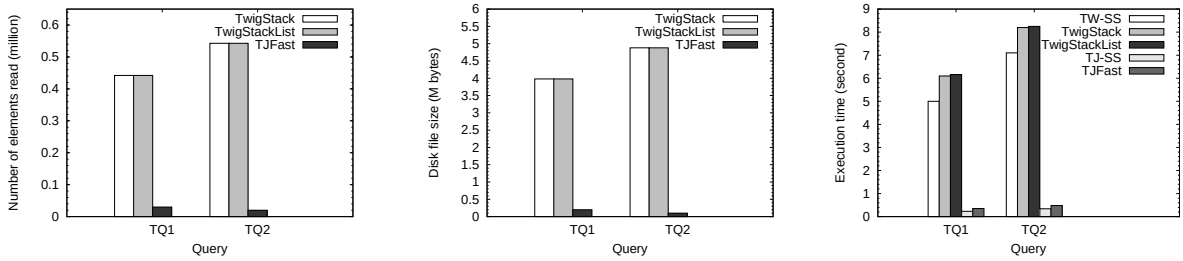
In order to research the effect of query path length on TJFast and PathStack, we then used the random data set consisting of ten different labels A_1,A_2,...,A_{10}, and issue path queries of different lengths such as $A_1/A_2/.../A_{10}$. Figure 9 shows the execution times of both techniques, as well as

[4]We choose these queries according to XMark benchmark queries in [18].

Table 3: Path Queries on XMark data

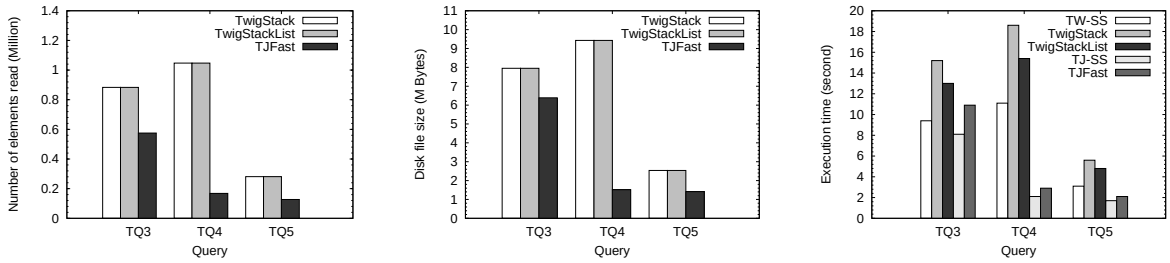
Path	Query
PQ_1	/site/closed_auctions/closed_auction/price
PQ_2	/site/regions//item /location
PQ_3	/site/people/person/gender
PQ_4	/site/open_auctions/open_auction/reserve

the number of elements read and the size of disk files. Clearly, TJFast results in considerably better performance than PathStack. The performance of PathStack degrades significantly with the increase of the path length, but that of TJFast is almost not affected at all.



(a)Number of elements scanned (b) Size of disk files scanned (c) Execution time

Figure 10: TwigStack,TwigStackList versus TJFast on DBLP

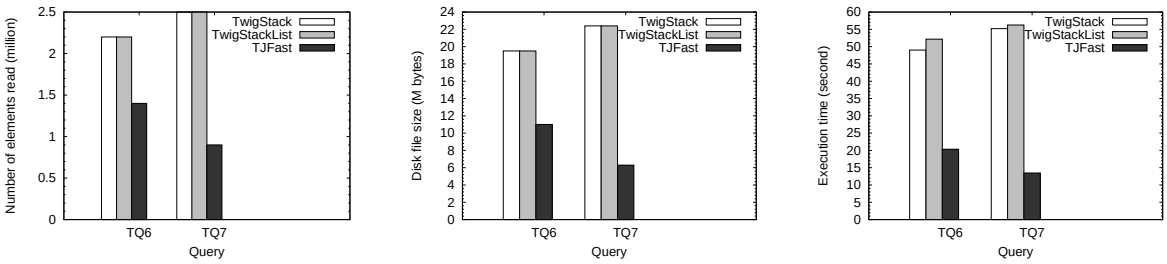


(a)Number of elements read (b) Size of disk files scanned (c) Execution time

Figure 11: TwigStack,TwigStackList, TJFast on TreeBank

5.2.2 Twig Queries

We now focus on twig queries, and compare four holistic twig join algorithms TwigStack, TwigStack-List, iTwigJoin and TJFast, We tested several XML queries on DBLP , TreeBank and XMark data(see



(a)Number of elements read (b) Size of disk files scanned (c) Execution time

Figure 12: TwigStack,TwigStackList, TJFast on XMark

Table 4: Twig Queries on DBLP and TreeBank(TB)

Twig	Data	Query
TQ_1	DBLP	//inproceedings//title[.//i]//sup
TQ_2	DBLP	//article[.//sup]//title//sub
TQ_3	TB	/S[.//VP/IN]//NP
TQ_4	TB	/S/VP/PP[IN]/NP/VBN
TQ_5	TB	//VP[DT]//PRP_DOLLAR_
TQ_6	XMark	//text[bold]/text//emph
TQ_7	XMark	//listitem[.//bold]/text[.//emph]/keyword

Table 4). These queries have different twig structures and combinations of parent-child and ancestor-descendant relationships. In particular, queries TQ1,TQ2 contain only ancestor-descendant relationships, while TQ4 contains only parent-child relationships. TQ3 contains only ancestor-descendant relationships between the branching node and its children, but TQ5,TQ6,TQ7 contains both parent-child and ancestor-descendant relationships to connect the branching node.

TJFast vs. TwigStack We first compare the performance between TJFast and TwigStack. From Figure 10 , 11 and 12, we see that TJFast outperforms TwigStack for all queries. We now analyze the query performance under two scenarios namely *the cost of disk access* and *the size of intermediate results.*

Cost of disk access Figure 10(a) and 11(a) ,12(a) show that TJFast read far fewer elements than TwigStack. For example, in TQ1, TwigStack read 442167 elements, but TJFast read only 2380 elements (over two orders of magnitude). This huge gap results from the fact that TwigStack scans

the elements for *all* nodes in the query, but TJFast scans only elements for *leaf* nodes. Figure 10(c) and 11(c) also show the elapsed time TwigStack and TJFast take to do a sequential scan over the input data (labels as TW-SS and TJ-SS, for TJFast, TJ-SS has included the time for decoding *extended Dewey* labels).

Size of intermediate results Table 5 shows the number of intermediate path solutions output by different algorithms. The last column is the number of intermediate solutions that contribute to final answers. An immediate observation is that TwigStack outputs many "*useless*" path solutions when query contains parent-child edges. For example, in TQ_3, TwigStack produced 702391 intermediate paths, while only 22565 are useful. More than 95% intermediate solutions output by TwigStack are "*useless*" to the final answers. Note that, unlike TwigStack, TJFast is optimal for queries TQ_3, since the number of paths produced by TJFast is 22565, which equals the number of useful solutions.

Table 5: Number of intermediate path solutions

Query	TwigStack	TwigStackList	TJFast	Useful
TQ_3	702391	22565	22565	22565
TQ_4	2237	388	388	302
TQ_5	10663	9	9	5

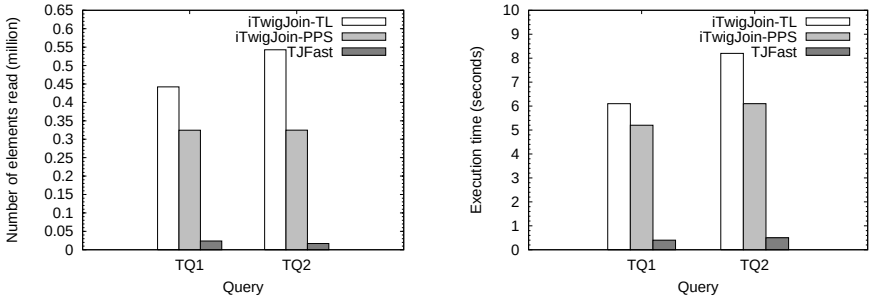
TJFast vs. TwigStackList For all queries, TJFast outperforms TwigStackList again (see Fig 10,11,12). This can be explained by the fact that TJFast reduces the I/O cost of TwigStackList by reading labels of only *leaf* nodes.

When queries contain parent-child relationships between the branching node and its children (i.e. queries TQ4,TQ5), both TwigStackList and TJFast are sub-optimal. Their sub-optimality is evident from the observation that the number of intermediate path solutions by TwigStackList and TJFast is slightly larger than the number of useful solutions.

TJFast vs. iTwigJoin We now compare the performance between TJFast, iTwigJoin and iTwigJoin is also based on region encoding, but it can be applied on different data partition strategies. Since [5] proposed two new data partition strategies (i.e. tag+level and PPS), we compare both with TJFast (labeled as iTwigJoin-TL and iTwigJoin-PPS, respectively).

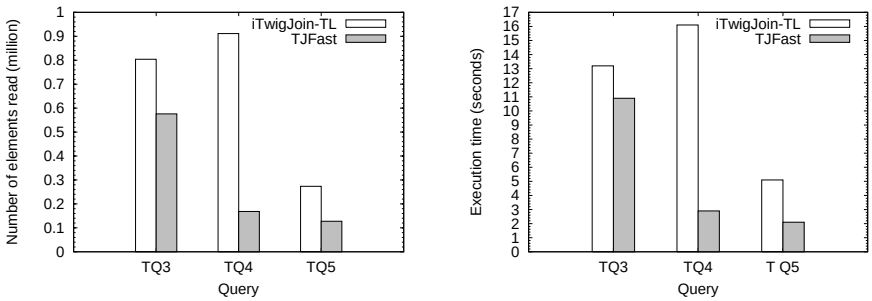
Performance results and the number of elements read for iTwigJoin-TL , iTwigJoin-PPS and

TJFast on DBLP and TreeBank data are shown in Figure 13 and 14,15. Since [5] has shown that PPS is not applicable to *deep* recursive data, for TreeBank, we only compared iTwigJoin-TL with TJFast. As shown in these figures, we can see that for all queries, TJFast is again more efficient than iTwigJoin-TL and iTwigJoin-PPS. Although iTwigJoin uses the refined data partition strategies and scan less elements than TwigStack and TwigStackList, the number of elements processed by iTwigJoin is still more than that by TJFast.



(a) # of elements read (b) Execution time

Figure 13: iTwigJoin,TJFast on DBLP

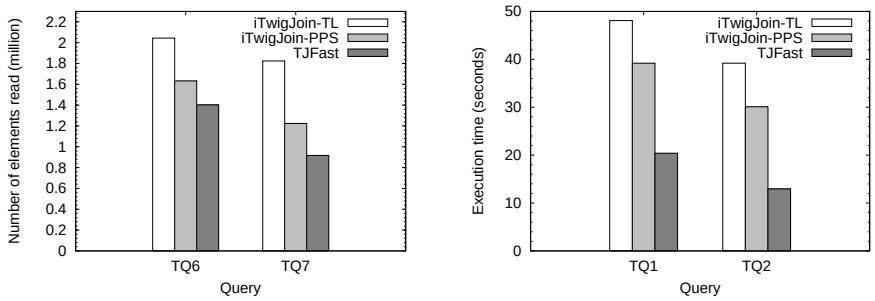


(a) # of elements read (b) Execution time

Figure 14: iTwigJoin,TJFast on TreeBank

5.2.3 Wildcards Query

Finally, we tested two wildcards queries Q1://NP[.//CD]/*/V, Q2://VP/*[PP-8]/PP-7 in TreeBank dataset. Q1 is a twig query consisting of a wildcard in a non-branching node, but Q2 is a branching wildcard twig query. For Q1, all four algorithms can be applied (with little modification for those algorithms on region encoding). But the performance of TJFast is much better than the best



(a) # of elements read (b) Execution time

Figure 15: iTwigJoin,TJFast on XMark

algorithms on region encoding[5](0.9s vs. 7.2s). For Q2, the family of algorithms on region encoding are significantly *affected* by wildcards in branching nodes, as they do not know which elements can be used to match this wildcard. Since there is no DTD available for TreeBank data, a brute-force solution is to access all elements to answer this query. Clearly, this method is unacceptably slow. In contrast, the existence of wildcard in branching nodes *does not affect* TJFast, which takes 0.3s to answer Q2. This shows that TJFast supports efficient processing of both non-branching as well as branching wildcard queries. Note that, in Q2, TwigStack, TwigStackList and iTwigJoin cannot scan only VP,PP-8,PP-7 to answer this query. This is because according to region encoding, even if PP-8 and PP-7 are descendants of VP and their level difference with VP is 2, PP-8 and PP-7 may not share the common parent.

Summary TJFast significantly outperforms TwigStack, TwigStackList and iTwigJoin under all settings (including shallow and deep documents, path and twig queries, branching and non-branching wildcards queries). The improvement is due to the facts that TJFast only scans labels for query *leaf* nodes. Algorithms on region encoding are comparable to TJFast only when the number of elements for all *internal* query nodes is very small.

6 Related work

Labeling schemes *Dewey ID* labeling scheme comes from the work of Tatarinov et al.[20] to represent XML order in the relational data model, and to show how this labeling scheme can

[5]In this case the best algorithm on region encoding is iTwigJoin-TL.

be used to preserve document order during XML query processing. O'Neil et al.[15] introduced a variation of prefix labeling scheme called ORDPATH. Unlike our *extended Dewey*, the main goal of ORDPATH is to gracefully handle insertion of XML nodes in the database.

The *region encoding* is considered as the work of Consens and Milo[7], who discuss a fragment of PAT text searching operators for indexing text database. Then Zhang et al[25] introduce it to XML query processing using inverted list. Recently, many researchers ([3],[19],[23]) have begun to design a dynamic XML labeling scheme on the context of frequent inserting and deleting data.

Twig join algorithms Al-Khalifa et al.[1] started the stack-based algorithms for XML structural joins. N. Bruno et al. [2] proposed a holistic twig join algorithm, namely TwigStack. Lu et al.[12] proposed TwigStackList, which identifies a larger optimal query class than TwigStack. Lu et al.[13] also researched how to answer an *ordered* twig pattern based on region encoding. Chen et al.[5] proposed an algorithm iTwigJoin, which is still based on region encoding. But unlike the previous work, iTwigJoin can be applied on different data partition strategies (e.g. Tag+Level and prefix Path Streaming).

Jiang et al. [9] proposed a general algorithm called TSGeneric+ based on indexes built on element labels. Their method can "jump" elements and achieve sub-linear performance for selective queries. But for evaluating queries with parent-child relationships, TSGeneric+ still may output many "useless" intermediate results like TwigStack. Jiang et al.[10] also studied the problem of processing queries with OR predicates. BLAS by Chen et al. [6] proposed a bi-labelling scheme: D-Label and P-Label for accelerating *parent-child* relationship processing. Their method decomposes a twig pattern into several *parent-child* path queries and then merges the results.

Yang et al. [24] proposed the idea of the combination of path index table and Dewey labels.[6] Similar to our TJFast, to answer a twig query, their method also can reduce I/O cost by accessing only the labels of leaf query nodes. But unlike TJFast, their algorithm did not fully explore the nice property of *Dewey* labels and only modified one procedure in TSGeneric+. So similar to TSGeneric+, their algorithm is still not efficient for processing queries with parent-child relationships.

ViST and PRIX ([22],[17]) transform both XML data and queries into sequences and answer XML queries through subsequence matching. While their methods avoid join operations in query

[6]Note that our work are developed independently of and differs considerably with [24].

processing, to eliminate false alarm and false dismissal, they resort to post-processing(for false alarm) and multiple isomorphism queries processing([21])(for false dismissal), both of which are time consuming.

Wildcards queries Chan et al. [4] used layer axis to minimize wildcards steps. Although their methods can replace wildcards by layer axis, in the scenario of XML twig pattern matching for a large document, the efficient evaluation of layer axis is still an issue.

7 Conclusions and Future Work

XML twig pattern matching is a key issue for XML query processing. In this paper, we have proposed TJFast as an efficient algorithm to address this problem using a novel labeling scheme: *extended Dewey*. Although the idea of *original Dewey* is not new, extending it to efficiently process XML twig pattern matching is nontrivial. This is because based on the *original Dewey*, we cannot know the element names along a path. To answer a twig query, we need to access the labels of *all* query nodes. Considering the fact that prefix comparison is less efficient than integer comparison, the performance of algorithm with the *original Dewey* is usually worse than that with *region encoding*. However, owing to our extension, *extended Dewey* has the important property: **Ancestor Name Vision**. So TJFast only needs to access labels of *leaf* nodes to answer queries and significantly reduce I/O cost. Further, TJFast can efficiently evaluate branching wildcards queries, which cannot be handled by algorithms with region encoding. As part of future work, we would like to improve *extended Dewey* to become an insert-friendly labeling scheme in the context of dynamic XML trees.

References

[1] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. of ICDE Conference*, pages 141–152, 2002.

[2] N. Bruno, D. Srivastava, and N. Koudas. Holistic twig joins: optimal XML pattern matching. In *SIGMOD Conference*, pages 310–321, 2002.

[3] B. Catania, B. C. Ooi, W. Wang, and X. Wang. Lazy xml updates: Laziness as a virtue of update and structural join efficiency. In *SIGMOD*, To appear 2005.

[4] C. Y. Chan, W. Fan, and Y. Zeng. Taming xpath queries by minimizing wildcard steps. In *Proceeding of VLDB*, pages 156–167, 2004.

[5] T. Chen, J. Lu, and T. W. Ling. On boosting holism in xml twig pattern matching using structural indexing techniques. In *SIGMOD*, pages 455–466, 2005.

[6] Y. Chen, S. B. Davidson, and Y. Zheng. BLAS: An efficient XPath processing system. In *Proc. of SIGMOD*, pages 47–58, 2004.

[7] M. P. Consens and T. Milo. Optimizing queries on files. In *SIGMOD*, pages 301–312, 1994.

[8] G. H. Gonnet. The PAT text searching sytem. Technical report, University of Waterloo, 1987.

[9] H. Jiang et al. Holistic twig joins on indexed XML documents. In *Proc. of VLDB*, pages 273–284, 2003.

[10] H. Jiang, H. Lu, and W. Wang. Efficient processing of XML twig queries with OR-predicates. In *Proc. of SIGMOD Conference*, pages 274–285, 2004.

[11] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proc. of VLDB*, pages 361–370, 2001.

[12] J. Lu, T. Chen, and T. W. Ling. Efficient processing of xml twig patterns with parent child edges: a look-ahead approach. In *CIKM*, pages 533–542, 2004.

[13] J. Lu, T. W. Ling, T. Yu, C. Li, and W. Ni. Efficient processing of ordered XML twig pattern matching. In *DEXA To appear*, 2005.

[14] U. of Washington XML Repository. http://www.cs.washington.edu/research/xmldatasets/.

[15] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-friendly XML node labels. In *SIGMOD*, pages 903–908, 2004.

[16] R. Y. Pinter. Efficient string matching with don't care patterns. In *Combinatorial ALgorithms on Words, NATO ASI Series*, volume 12, pages 11–29, 1985.

[17] P. Rao and B. Moon. PRIX: Indexing and querying XML using prufer sequences. In *ICDE*, pages 288–300, 2004.

[18] A. R. Schmidt et al. Xmark an xml benchmark project. http://monetdb.cwi.nl/xml/index.html.

[19] A. Silberstein, H. H. nd K. Yi, and J. Yang. Boxes: Efficient maintenance of order-based labeling for dynamic XML data. In *Proc. of ICDE.*, pages 285–296, 2005.

[20] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang:. Storing and querying ordered XML using a relational database system. In *Proc. of SIGMOD*, pages 204–215, 2002.

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

[22] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A dynamic index method for querying XML data by tree structures. In *SIGMOD*, pages 110–121, 2003.

[23] X. Wu, M. Lee, and W. Hsu. A prime number labeling scheme for dynamic ordered XML trees. In *Proc. of ICDE*, pages 66–78, 2004.

[24] B. Yang, M. Fontoura, E. J. Shekita, S. Rajagopalan, and K. S. Beyer. Virtual cursors for XML joins. In *CIKM*, pages 523–532, 2004.

[25] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *Proc. of SIGMOD Conference*, pages 425–436, 2001.