T H E N A T I O N A L U N I V E R S I T Y
of S I N G A P O R E

S c h o o l o f C o m p u t i n g
Lower Kent Ridge Road, Singapore 119260

TR12/04

TJFast: Efficient Processing of XML Twig Pattern Matching

Jiaheng LU, Ting CHEN and Tok Wang LING

November 2004

TechnicalReport

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

TJFast: Efficient Processing of XML Twig Pattern Matching

Jiaheng Lu , Ting Chen, Tok Wang Ling
School of Computing, National University of Singapore
{lujiahen,chent,lingtw}@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. Previous twig join algorithms were proposed as an I/O optimal solution when the twig pattern only involves ancestor-descendant relationships. In this paper, we propose a new efficient holistic twig join algorithm, namely TJFast, which is based on a variation of *Dewey ID* labeling scheme. In order to answer a twig query, TJFast only needs to access the labels of elements whose tags appear in the *leaf* nodes of query. TJFast guarantees the I/O optimality for queries with only *ancestor-descendant* edges connecting branching nodes and their child nodes. In other words, unlike previous algorithms, the optimality of TJFast allows *parent-child* edges to appear in some edges. Besides TJFast, we also present a novel streaming strategy, called *tag+level* streaming, where two elements appear in the same stream if and only if they have the same *tag name* and *level number*. We develop a twig join algorithm, called TJFast$^+$ based on *tag+level* streaming. We show that TJFast$^+$ can identify even larger query class to guarantee I/O optimality than TJFast. Therefore, in this paper, by proposing TJFast/TJFast$^+$, we significantly enlarge the optimal query class compared to previous algorithms. 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. A variety of languages (e.g. XPath, XQuery) have been proposed for this purpose, all of which can be viewed as consisting of a pattern language and navigation expressions. Since the

data objects are typically trees, *twig* (i.e. a small tree) pattern matching is the central issue.

In practice, XML data may be very large, complex and have deeply 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 ([3, 8, 9, 10, 21]) 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. pre- and postorder [5], extended preorder [14]) or textual positions of *start* and *end* tags (e.g. region encoding [3], BOXes [18]) or path expressions(e.g. Dewey ID [20], ORDPATH [17]). 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 whether they have ancestor-descendant (or parent-child) relationship, 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 a sample XML document. Given the label "1.9.2.2" 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 predicate in the query. Further, this feature enables us to easily match simple path pattern by string matching. Take element "1.9.2.2" 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 pattern (e.g. "*//book//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 al-

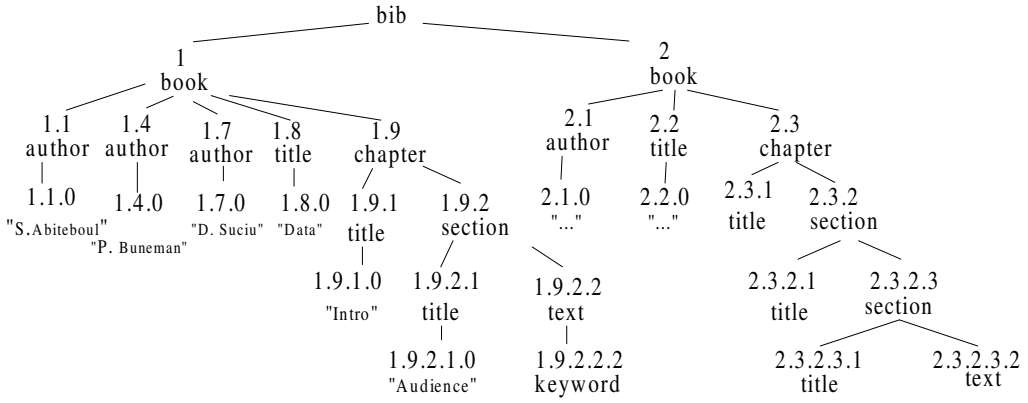


Figure 1: A sample XML tree

gorithms have been developed to efficiently process twig queries. In particular, Al-Khalifa et al. [1] propose to decompose the twig pattern into many binary relationships, then use Tree-merge or Stack-tree algorithm to match the binary relationships. The main disadvantage of their approach is that the intermediate results may be very large. Bruno et al. [3] propose pathStack/TwigStack. They use a chain of linked *stacks* to compactly represent partial results to the root-leaf query paths. For evaluating queries with only *ancestor-descendant* edges, TwigStack guarantees that each intermediate path solution contributes to final answers. While TwigStack demonstrates the superiority compared to binary decomposition-based approach, such an algorithm has a limitation: the size of "*useless*" intermediate path solutions still cannot be effectively controlled when query contains any *parent-child* edge. As a result, a natural question that follows is "Can we develop a more efficient algorithm than TwigStack to reduce the size of intermediate path solutions when the query contains some parent-child edges?"

The answer is certainly yes. In this paper, based on the *extended Dewey*, we present a new efficient algorithm, namely TJFast(i.e. a Fast Twig Join algorithm), which is I/O optimal for a wider class of queries than TwigStack. In particular, if we call edges that connect a branching node and its children as *branching* edges, then when queries contain only ancestor-descendant relationships in *branching* edges, the I/O cost of TJFast is only proportional to the sum of sizes of the input and the final output. In other words, unlike TwigStack, TJFast guarantees the I/O optimality when *parent-child* relationships exist in *non-branching* edges. Furthermore, even in the case when there are parent-child relationships in *branching* edges, we show that TJFast typically produces significantly smaller intermediate results than TwigStack. These improved results mainly owe to *extended Dewey*

labeling scheme, since it broadens our outlook to enable us to see the *whole path* as we read the label of a *single* element.

In addition to the *extended Dewey* and TJFast, we make the contribution by proposing a new streaming method. Previous join algorithms cluster elements to the streams according to their element names alone. In this paper, we present a *tag+level* streaming strategy, where two elements appear in the same stream if and only if they have the same *tag* and *level number*. *Tag+level* streaming is a refinement data partition strategy of the previous one. We demonstrate two significant advantages of *tag+level* streaming as follows.

(1) Before reading input data, according to the level information, we can prune away some streams in which elements do not participate in answers. Thus, we skip elements and reduce the cost of disk access.

(2) Based on tag+level streaming, we propose a new holistic twig join algorithm, called TJFast$^+$, which shows I/O optimality for a broader query class than TJFast.

Figure 2. summarizes the optimal query class of three algorithms. From this figure, we see that TJFast$^+$ shows the largest one, which covers all three twig patterns in Figure 2. The next one is TJFast, which cover Q1 and Q2. The smallest one is TwigStack, which includes only Q1 .

The outline and contribution of this paper are summarized as follows.

Section 3 : we present a new labeling scheme, called *extended Dewey ID*, which provides an *extraordinary* chance for us to design an efficient twig join algorithm.

Section 4: we present a new holistic twig join algorithm, namely TJFast, which identifies a larger class of queries to show I/O optimality than TwigStack.

Section 5: we present a new streaming approach: *tag+level* and develop an algorithm TJFast$^+$ based on this streaming. We analytically show that TJFast$^+$ achieves better performance than TJFast by the reduction of disk access and the enlargement of optimal query class.

Section 6: Experimental results on a variety of queries and data sets are presented and analyzed. These experimental results validate our analytical results and demonstrate the significantly superiority of our algorithms over the previous one.

We close this paper by discussing related work and future work.

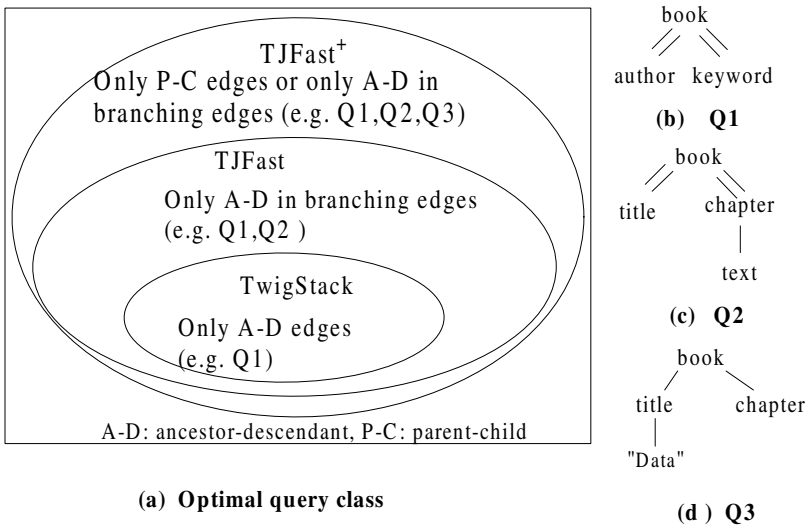


Figure 2: Optimal query class for three algorithms

2 Preliminary

2.1 Data model and XML twig pattern

We model XML documents as *ordered* trees, where nodes represent elements, attributes and string values (CDATA, PCDATA), and parent-child pairs represent nesting between XML element nodes. 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, an attribute, a wildcard(*) symbol or a string value. The query twig pattern edges are either parent-child or ancestor-descendant edges. If a node in the query twig has more than one child, it is a *branching* node. Further, a *branching* edge is defined as an edge between a branching node and its child nodes.

Given a twig pattern T and an XML database D, a match of T in D is identified by a mapping from the nodes in T to the elements in D, such that: (i) the query node predicates are satisfied by the corresponding database elements (wherein wildcard "*" matches 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. This labeling scheme supports efficient evaluation of structural relationships between elements. Formally, 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 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 all ancestors of an element 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" presents the path "a/b/c" from the root to u, then it is quite straightforward to identify whether the elements along this path matches a simple path pattern (e.g. "a//c"). Inspired by this observation, we develop an *extended* Dewey ID labeling scheme which provides an *extraordinary* chance for us to design a new approach to match XML path (and twig) pattern.

3 Extended Dewey and FST

In this section, we first introduce the *extended Dewey* labeling scheme, where we encodes the names of elements along a path into a single label. Then we present a Finite State Transducer(FST) to derive 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.

3.1 Extended Dewey

The intuition of *extended Dewey* is to use *module* 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 Dewe*y, we need to know a little additional schema information, which we call a *schema clue*. In particular, given any tag t in a document, the *schema clue* is all possible (distinct) names of children of elements with name t. This clue is easily derived from DTD, XML schema or statistic data on the document. Let us use $CT(t) = \{t_1, t_2, \cdots, t_n\}$ to denote the *schema clue* of tag t. Suppose

```
<!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 3: DTD for XML document in Fig 1

there is an ordering for tags in $CT(t)$, where the particular ordering is not important. For example, consider the DTD in Figure 3; the tags of all possible children of *book* are *author*, *title* and *chapter*. So $CT(book) = \{author, title, chapter\}$. Using schema clue, we may easily create a mapping from an integer to an element name (i.e. element tag). Suppose $CT(t) = \{t_1, t_2, \cdots, t_n\}$, for any element e_i with name t_i, if $i \neq n$. we assign an integer x_i to e_i such that $x_i \ mod \ n = i$, otherwise, $x_i \ mod \ n = 0$. 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 *title*, 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 in an XML tree,

(1) if u is a text value , then $x = 0$;

(2) otherwise, assume that the element name of u is the k-th tag in $CT(t_s)$, 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, then

$$x = \begin{cases} \left\lfloor \frac{y}{|CT(t_s)|} \right\rfloor \cdot |CT(t_s)| + k & w < k; \\ \\ \left\lceil \frac{y}{|CT(t_s)|} \right\rceil \cdot |CT(t_s)| + k & w \geq k. \end{cases}$$

where $|CT(t_s)|$ denotes the size of $CT(t_s)$ and if (y mod $|CT(t_s)|$)=0, then $w = |CT(t_s)|$

, else $w = y \bmod |CT(t_s)|$.

EXAMPLE 3.1 Figure 1 illustrates an XML document tree that conforms to the DTD in Figure 3 and the *extended Dewey* label of each element. For instance, the label of *chapter* under *book(1)* is computed as follows. Here $k = 3$ (for *chapter* is the third tag in its schema clue), y=8 (for the last component of "1.8" is 8), $|CT(book)| = 3$, so $w = y \ mod \ 3 = 2$. Thus $w < k$. Then $x = \lfloor 8/3 \rfloor * 3 + 3 = 9$. So the *chapter* is assigned "1.9".

Note that the *extended Dewey* label also allows for checking order (e.g. in Figure 1, the element *chapter*(1.9) follows the element *author*(1.7), for $9 > 7$).

Index size Comparing the *original* and *extended Dewey* labeling scheme, astute readers may find that *extended Dewey* does not assign integers consecutively; it often skips some integers. For example, in Figure 1, the first *author* is labeled 1.1, but the second *author* is 1.4. We skip 1.2 and 1.3. Of course, as mentioned above, the purpose of such skipping is to create a mapping from integers to element names. But does this skipping significantly increase the size of labels? Next we shall show that the *extended Dewey* does not alter the asymptotic space complexity of the original one.

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 sub-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 an XML document, the *extended Dewey* labels can be efficiently generated by scanning the document once. Then during twig pattern matching, we use a *finite state transducer* (FST) to convert the label of each element into the sequence of element names which reveals the *whole path from the root to this element*. We begin this section by presenting Function $F(t, x)$ which will be used to define FST.

Definition 1. Suppose N denotes natural number set and Σ denotes the definite alphabet of all distinct tag names in an XML document T. Given an tag t in T, $CT(t) = \{t_1, t_2, \cdots, t_n\}$, a function $F(t, x)$: $\Sigma \times N \rightarrow \Sigma$ can be defined by $F(t, x) = t_k$, where

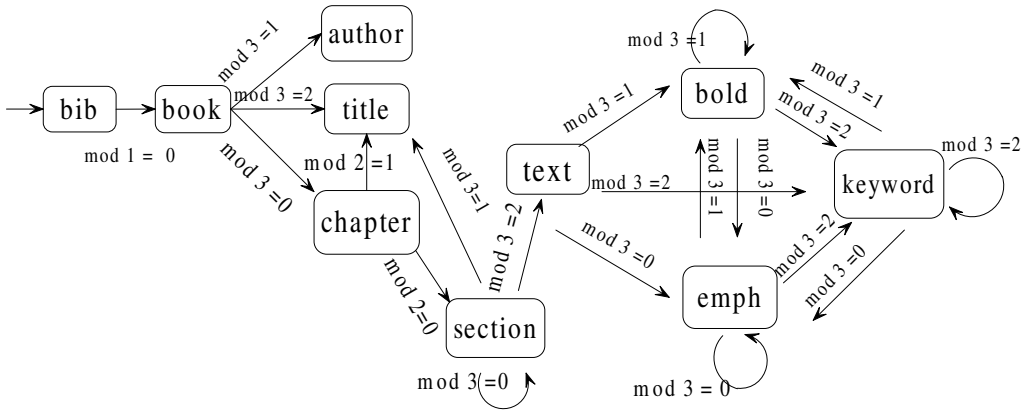


Figure 4: A sample FST for DTD in Fig 3

$$k = \begin{cases} |CT(t)| & \text{if x mod } |CT(t)| = 0; \\ \text{x mod } |CT(t)| & \text{otherwise.} \end{cases}$$

Definition 2. (**Finite State Transducer**) Given *schema clues* and an *extended Dewey* label, we can use a *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 $I = N \cup \{0\}$; (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 $x = 0$, $\delta(t, x) = PCDATA$, 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 Fig 4 gives an FST for the DTD in Fig 3. For clarity, we do not draw the state PCDATA here. If the current input number is 0, any state directly transits to PCDATA and terminates. Suppose the input sequence is "1.9.2.2.2". By using FST, the output sequence is "*bib.book.chapter.section.text.keyword*", which shows the path from *bib* to *keyword* in the document. □

The function of FST can be summarized as follows. Given an extended Dewey $label(u)$,(i)if the last component of $label(u)$ is 0, then u corresponds to the text value of an element, say element s. FST outputs the element names along the path from the *root* to s; (ii)otherwise, u corresponds to an element. Then FST outputs the element names along the path from the *root* to u.

4 Twig Pattern Matching

4.1 Path matching algorithm

It is straightforward to evaluate a query path pattern in our approach. We only need to scan the elements whose tags appear in *leaf* nodes of query. For each visited element, we first use FST to convert its label into element names along the path from the root to it, and then perform string-matching against it[1]. If the path from the root to this element matches the desired path pattern, then we directly output the matching answers. As a result, we evaluate the path pattern efficiently by scanning the input list once and ensure that each path solution is our desired final answer.

It is worth to notice that the I/O cost of our approach is typically much smaller than that of previous path pattern matching algorithms (e.g. PathStack [3]), for we scan only one input list for the query *leaf* node, but they need to scan lists for *all* query nodes.

4.2 Twig matching algorithm

Based on *extended Dewey* labels, this section presents a holistic twig pattern join algorithm, which we call TJFast. We will first introduce some data structures and notations to be used in TJFast.

4.2.1 Data Structures and Notations

Let q denote a twig pattern and p_f denote a path pattern from the *root* to the *leaf* node f. In our algorithms, we make use of the following twig node operations: `isLeaf`: $Node \rightarrow Bool$, `leafNodes`: $Node \rightarrow \{Node\}$, `children`: $Node \rightarrow \{Node\}$. The result of `leafNodes(q)` is all leaf nodes in the twig q. In the rest of the paper, "node" always refers to a tree node in the twig pattern (e.g. we often use f to denote a leaf node), while "element" refers to the elements in the dataset involved in a twig join. (e.g. element e). 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 predicate f. The elements in the stream are sorted by the ascending lexicography order. For example, "1.2" precedes "1.3" and "1.3" precedes "1.3.1". The operations over streams are eof, *advance*, *get*. The last operation *get* returns the *extended Dewey* label of the current element in the stream. Initially, $get(T_f)$ returns the first elements of stream T_f. Function $end(T_f)$ tests whether $get(T_f)$ is at the end of T_f.

[1]There are a rich set of literature on efficient string processing without or with wildcards e.g. [19].

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 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 contribute to final answers. Set S_b is initially empty.

Algorithm 1 TJFast(q)

Input: q is a query twig pattern

1: **for all** $f \in$ leafNodes(q) **do**
2: locatedMatchedLabel(f)
3: **end for**
4: **while** ($\neg end(q)$) **do**
5: f_{act} = getNext($root$)
6: outputSolutions(f_{act})
7: advance(f_{act})
8: locatedMatchedLabel(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 $end(q)$

1: Return $\forall f \in leafNodes(q) \rightarrow end(T_f)$

Procedure outputSolutions(f)

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

4.3 TJFast

Algorithm TJFast, which computes answers to a query twig pattern q, is presented in Algorithm 1 and the procedure *getNext* is shown in Algorithm 2. TJFast operates in two phases. In the first phase (line 1-9), some solutions to individual root-leaf path pattern are

Algorithm 2 getNext(n)

1: **if** (isLeaf(n)) **then**
2: return n
3: **else if** (n has only one child) **then**
4: return $getNext(child(n))$
5: **else**
6: **for** $n_i \in$ children(n) **do**
7: $f_i = getNext(n_i)$
8: $e_i = max\{p | p \in \text{MatchedPrefixes}(f_i, n)\}$
9: **end for**
10: max $= maxarg_i\{e_i\}$
11: min $= minarg_i\{e_i\}$
12: **for all** $e \in$ MatchedPrefixes(f_{min}, n) **do**
13: **if** (e is a prefix of e_{max}) **then**
14: moveToSet(S_n, e)
15: **end if**
16: **end for**
17: return n_{min}
18: **end if**

Function MatchedPrefixes(f, b)

1: Return a set of element p that is an ancestor of $get(T_f)$ such that p can match node b in the path solution of $get(T_f)$

Procedure moveToSet(S, e)

1: Delete any element in S that has not ancestor-descendent(or parent-child) relationship with e
2: Add e to set S_b

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, it is easy to identify whether the path from the root to this element matches the corresponding path pattern. Thus, the key problem of TJFast is to determine whether a path solution possibly contributes to final answers. 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 node in the query. 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 tries to find the highest possible elements that likely match the branching nodes in the final solution and cache them in set S_b.

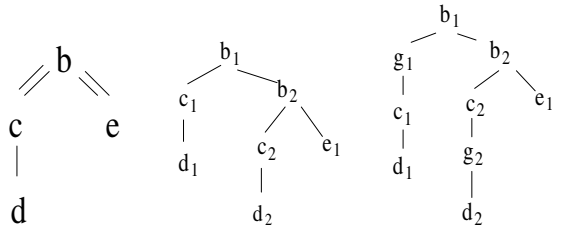
The main procedure of TJFast is not difficult to be understood(see Algorithm 1). In line 1-3, for each stream, we identify the first element whose path matches the individual path pattern. In line 5, we identify the stream $T_{f_{act}}$ to be processed by using *getNext(root)*. Given the knowledge of elements in sets, some path solutions for the current element in $T_{f_{act}}$ are output in line 6, Then we advance $T_{f_{act}}$ (line 7) and locate the next matching element in $T_{f_{act}}$ (line 8).

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 cache appropriate elements to sets, as discussed follows.

Algorithm *getNext(n)* returns a query leaf node f, according to the following recursive criteria (i) if n is a leaf node, $f = n$(line 2) ; else (ii) if n is not a branching node, $f = getNext(child(n))$(line 4); else (iii) n is a branching node. For each stream $f_i = getNext(n_i)$, $n_i \in children(n)$, suppose element e_i matches node n in the corresponding path solution(if more than one element that match n, e_i is the *deepest* one by level)(line 7,8), we return f_{min} such that the current element e_{min} in stream f_{min} has the minimal label in all e_i by lexicography order(line 11,17).

Now we discuss the criteria of inserting an element e_b to set S_b. This is important, since in line 6 of TJFast, we only output an individual path solution where each element that matches branching node must be found in the corresponding set. An element e_b in a path p of the dataset is called as "deepest" element to match node b if there are more than one elements to match b in p and e_b is a descendant (or child) of all other matching

elements. In *getNext*, before an element e_b is inserted to the set S_b, we ensures that e_b is an ancestor(or parent) of each other "deepest" element to match node b in all relative path solutions. Therefore, when the query twig pattern contains only ancestor-descendant relationships in all *branching* edges, element e_b is guaranteed to match branching node b in each path solution. This ensures that each individual path solution is merge-joinable with at least one solution to all other query paths, as illustrated in follows.



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

Figure 5: Illustration to TJFast

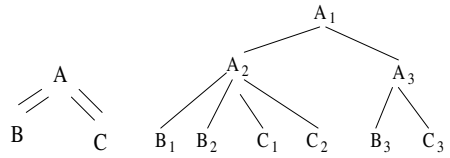
EXAMPLE 4.1 Consider the query Q1 and Doc1 in Fig 5(a),(b). A subscript is added to each element in the order of pre-order traversal for easy reference. There are two input streams T_d and T_e. Initially, the two elements are d_1 and e_1. Observe that b_1 matches b in the solution of $b_1/c_1/d_1$ to pattern $b//c/d$ and b_2 is the *deepest* element to match node b in the solution of $b_1/b_2/e_1$ to pattern $b//e$. Since b_1 is the *parent* of b_2, we insert b_1 to set S_b and output path solutions $<b_1, c_1, d_1>$ and $<b_1, e_1>$. Next we advance T_d and scan d_2. Now it is easy to verify that the deepest elements to match node b in streams T_d and T_e are b_2. Thus, b_2 is inserted to set and $<b_2, c_2, d_2>$,$<b_2, e_1>$ are output. Finally, four path solutions are merged to form final twig matches. □

The second phase of Algorithm 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 ([2]). The main procedures of TJFast do not be changed, but we need to delay some outputs until no answers prior to them can be computed. The details of how to achieve this naturally in the scenario of TJFast can be found in the next section.

Remark Now we use an example to compare TJFast to TwigStack. Consider Q1 and Doc2 in Fig 5(a) and (c). TJFast does not insert b_2 to set S_b, since by reading the label of d_2, we immediately identify that d_2 does not contribute to final answers. In contrast, TwigStack pushes b_2 to *stack* S_b and output a "*useless*" intermediate path solution $<b_2, e_1>$. The

behavior of TwigStack is understandable because based on *region coding* of d_2, one cannot know whether d_2 has parent tagged with c. But based on *extended Dewey*, one can easily see that the parent of d_2 is tagged with g rather than c. This example shows the benefit of *extended Dewey* labeling scheme on efficiently controlling the size of intermediate results.

4.4 Block technique



(a)Query (b) Document

Figure 6: An example of XML data that need blocking

Consider the simple query and dataset in Fig 6 (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 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 immediately output path solutions for any element, but add them to the Self-list of its corresponding nearest branching node. For example, in Fig 6, 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 7):

(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)	(b)	(c)
$\{C_1, C_2, ..., C_n\}$ C is a branching node	$\{P_1, P_2, ..., P_m\}$ $\{C_1\}$ Both C and P are branching nodes and P is the nearest ancestor node of C in the query	$\{C_1\}$ 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 7: Possible stack configuration when blocking results

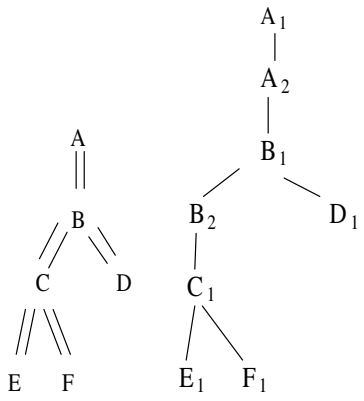
this case, we append the Self-list and Inherit-list of C_1 to the Self-list of each element P_i , where P_i is an ancestor of C_1.

(c) Node C has no ancestor that is a branching node in query(i.e. C is the top branching node). 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 and leaf 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 6 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 7(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 deleted from the set, all path solutions in the Self- and Inherit-lists of A_1 are output.

EXAMPLE 4.3 Consider the query and data set in Fig 8. 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



(a)Query (b) Document

Figure 8: Illustration to blocking

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.

4.5 Analysis of TJFast

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

Lemma 4.1. *In Procedure moveToSet 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}, label(e) < label(e_{new}) by lexicography order. Note that $a<b$ holds and a is not a prefix of b, then whatever postfix c, d is appended 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 4.2. *In line 13 of Function getNext, if element* e *is not a prefix of* e_{max} *and* e *does not appear in set* 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 firs case, according to the inductive hypothesis, element e does not participate in any final solution. The second case is impossible, since by Lemma 4.1, each deletion operation is *safe*. Therefore, the lemma is proved.

□

Lemma 4.1 shows that any element deleted from sets does not participate new solutions, so the deletion is *safe*. Lemma 4.2 shows that all elements that match branching nodes and involve in final answers are inserted to the corresponding sets, so the insertion is *complete*. These two lemmas are important to establish the correctness of the following theorem.

Theorem 4.3. *Given a twig query q and an XML database D, Algorithm TJFast correctly returns all answers for q on D.*

Proof. In Algorithm TJFast, we repeatedly call *getNext* to retrieve the next element to be processed(line 5). Then we output its path solutions in line 6 so that each element matching branching node can be found in the corresponding set. We know that the deletion and insertion in each set are *safe* and *complete* by Lemma 4.1 and 4.2. Therefore each path solution that contributes to final answer is guaranteed to be output in line 6. Finally, in the second phrase of TJFast, those path solutions are correctly merged to form the final answers. □

While the correctness holds for query with parent-child and ancestor-descendant edges, the I/O optimality holds only for the case where the twig query has only ancestor-descendant relationships in *branching* edges.

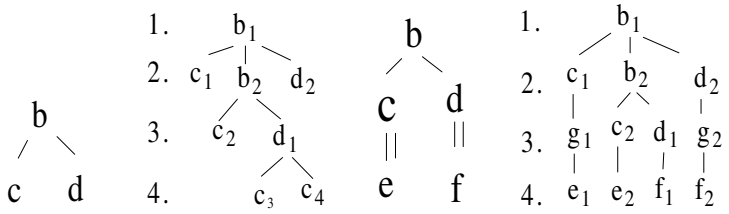
Theorem 4.4. *Consider an XML database D and a twig query q with only ancestor-descendant relationships in branching edges. The worst case I/O complexity of TJFast is linear to the sum of the sizes of input and output lists. The worst-case space complexity of*

this algorithm is that the number of leaf nodes in q times the length of the longest path in D.

Proof. The space complexity is easy to derive according to line 14 in Function *getNext*. Here we focus on the proof of I/O optimality. The following observation is important to prove the optimality of TJFast: when all branching edges are only *ancestor-descendant* relationships, in line 13 of *getNext*, since e is a prefix of e_{max}, we know that $e \in \text{MatchedPrefixes}(f_i, n)$ for each $f_i = getNext(n_i)$, where $n_i \in children(n)$. That is, e is guaranteed to be a common element in the path solutions of each stream T_{f_i}. Based on this observation, when all branching edges are only ancestor-descendant relationships, each element e that is inserted to set is guaranteed to appear in the path solutions of each T_{f_i}. Finally, note that we only output path solution, in which elements that match branching node occur in the corresponding set(line 6). Therefore, TJFast is I/O optimal when the query contains only ancestor-descendant relationships in branching edges. □

It is worth to note two advantages of TJFast over the previous algorithm TwigStack: (1) TJFast only scans the elements which satisfy the predicates of query *leaf* nodes, but Twigstack must scan elements for *all* nodes. So TJFast typically reduces disk access compared to TwigStack; (2) TJFast show a larger query class to guarantee the I/O optimality than TwigStack.

4.5.1 Sub-optimality of TJFast



(a)Q1 (b) Doc1 (c) Q2 (d) Doc2

Figure 9: Sub-optimality of TJFast and TJFast$^+$

Theorem 4.4 holds only for query with ancestor-descendant relationships in *branching* edges. Unfortunately, in the case where the query contains a parent-child relationships in *branching* edges, Algorithm TJFast is no longer guaranteed to be I/O optimal. In particular, the algorithm might produce a solution for one root-leaf path that does not contribute to final answers.

For example, consider the query $Q1$ and document $Doc1$ in Figure 9 ($Q2$ and $Doc2$ will be explained in the next section). There are two input streams in $Doc1$ and their first elements are c_1 and d_1 respectively. In this case, we cannot say any of them involves in a solution without advancing other stream, and we cannot advance any stream before knowing if it participates in a solution. Thus, optimality can no longer be guaranteed. In the next section, we will propose a novel algorithm TJFast$^+$, which uses a new streaming strategy and guarantees to be optimal for query $Q1$.

5 Twig Join on tag+level

As we already pointed out in previous section, TJFast is not optimal for some queries. In this section, we address this problem using a novel approach. Instead of clustering elements by their names alone, we put two elements in one stream if and only if they have the same *element names* and *level numbers*, which is called *tag+level* streaming. Based on *tag+level* streaming, we develop an algorithm, called TJFast$^+$, which can guarantee the I/O optimality for a larger class than TJFast.

5.1 Stream pruning

In the *tag+level* streaming, two elements appear in the same stream if and only if they have the same *element names* and *level numbers*. Thus, *tag+level* streaming is a refinement of data partition for the previous *tag* streaming. For example, consider the document Doc1 in Figure 9(b) again. Based on *tag+level* streaming, there are five input streams: $(c, 2)$,$(c, 3)$,$(c, 4)$, $(d, 2)$,$(d, 3)$, where the first field of each tuple denotes tag name and the second denotes level number. To answer a twig query, before structural join, we may explore the advantage of *tag+level* streaming and prune away those streams in which all elements do not involve in the solutions. Intuitively, given the knowledge of the level number, we may safely skip streams that cannot find any *matching* stream. For example, in Q1 of Figure 9, observe that both $<b, c>$ and $<b, d>$ are parent-child relationships. Therefore, in $Doc1$,we may safely skip all elements in stream $(c, 4)$ since there is not $(d, 4)$ stream at all. Algorithm 3 formalizes this pruning process. This is a two-way pruning algorithm. In method *pruneParent* , we use *bottom-up* way to prune the levels of parent nodes in queries according to their child levels. Then, in method *pruneChildren*, we use *top-down* way to prune the levels of child nodes according to their parent levels.

Generally speaking, when the twig query contains more parent-child edges, the effect of

stream pruning is more significant. This is because parent-child edges strictly specify the level difference between parent and child nodes. But with ancestor-descendant edges, we only require the level number of descendant node to be greater than that of ancestor node. So both ancestor and descendant nodes have many choices for their levels, which results in that few of streams can be pruned.

Algorithm 3 Stream pruning

/* Assume that at the beginning of program, $levels(n)$ is a set to contain all level numbers of tag n. At the end, $levels(n)$ contains only level numbers that likely contribute to query answers.*/

```
1: pruneParent(root)
2: pruneChildren(root)
```

Procedure pruneParent(n)

```
1: if isLeaf(n) then return
2: for n_i ∈ childNodes(n) do
3:    pruneParent(n_i)
4:    if (n_i, n) is a parent-child edge  then
5:       delete elements in levels(n) that cannot find child level in levels(n_i)
6:    else
7:       delete elements in levels(n) that cannot find descendant level in levels(n_i)
8:    end if
9: end for
```

Procedure pruneChildren(n)

```
1: if isLeaf(n) then return
2: for n_i ∈ childNodes(n) do
3:    if (n_i, n) is a parent-child edge  then
4:       delete elements in levels(n_i) that cannot find parent level in levels(n)
5:    else
6:       delete elements in levels(n_i) that cannot find ancestor level in levels(n)
7:    end if
8:    pruneChildren(n_i)
9: end for
```

Algorithm 4 getNext of TJFast$^+$

/* This following algorithm is used to evaluate query with only parent-child edge. Here only Function *getNext* is different to that in TJFast. */

Function getNext(n)

1: **if** (isLeaf(n)) **then**
2: return n
3: **else if** (n has only one child) **then**
4: return $getNext(child(n))$
5: **else**
6: **for** $n_i \in$ children(n) **do**
7: $f_i = getNext(n_i)$
8: $e_i =$ MatchedPrefixes(f_i, n)
9: **end for**
10: $max = maxarg_i\{e_i\}$
11: $min = minarg_i\{e_i\}$
12: **if** ($e_{min} == e_{max}$) **then**
13: moveToSet(S_n, e_{min})
14: **end if**
15: return n_{min}
16: **end if**

5.2 TJFast+

Now we present algorithm TJFast$^+$ to evaluate a twig pattern query based on *tag+level* streaming. The idea behind TJFast$^+$ is to use two procedures to evaluate queries based on whether the query contains ancestor-descendant edges. In particular, (i) when query contains only parent-child edges, we need to modify the previous TJFast so that the new algorithm guarantees I/O optimality in this case; and (ii) when query contains both parent-child and ancestor-descendant edges, as we will show in Section 5.4, it is very difficult to design an optimal algorithm. Therefore, we consider to reuse TJFast.

5.2.1 Query with parent-child

Based on *tag+level* streaming, it is feasible to design a twig join algorithm which is optimal for query with only parent-child edges. For this purpose, we modify TJFast to guarantee its optimality. Before evaluating the query, we first cluster streams to several *matching groups* according to their level numbers so that only elements in the same group possibly match the twig query. For example, consider *Doc*1 in Figure 9 again. There are two groups which likely match the query. One is $(c, 2)$,$(d, 2)$ streams, and the other is $(c, 3)$ and $(d, 3)$. But, for example, the combination of $(c, 2)$,$(d, 3)$ unlikely provides any query solution.

Algorithm 4 shows the modified *getNext* algorithm, which is used to answer query with only parent-child edges. The other parts of TJFast$^+$ is the same as TJFast. The input data is the streams within one *matching group*. We assume that elements in each stream are still in descending order of their *extended Dewey* labels. The only changes are in the lines 8,12. In line 8, since there is definitely one element to match branching node n, we do not use max function and directly let e_i = MatchedPrefixes(f_i, n). In line 12, $e_{min} = e_{max}$ means all e_i are the same element. Recall that in the similar context, Algorithm TJFast can only ensure that all e_i have ancestor-descendant relationships, which causes its sub-optimality for some queries. But here since all e_i are identical, e_{min} matches branching node n in the path solutions of each stream T_{f_i}. Therefore, Algorithm 4 guarantees that each path solution contributes to final answers.

5.2.2 Query with parent-child and ancestor-descendant

Let us now consider how to evaluate a query with both parent-child and ancestor-descendant edges based on *tag+level* streaming. Intuitively, we hope to reuse the TJFast algorithm. But here our main problem is that for each tag name, there are typically more than one input

streams so that TJFast cannot be directly applied. In order to solve this problem, we propose to *simulate* multiple streams with the same tag name to one *sorted* stream. In particular, our idea is that during structural join, we always retrieve the *minimal* element in all streams with the same tag name so that multiple streams work like a single *sorted* stream. This can be done by storing the head element of each stream in the main memory and maintain a *min-heap* data structure to efficiently retrieve their minimal one. Therefore, TJFast can be reused to evaluate query in this new scenario. Note that when query contains both ancestor-descendant and parent-child relationships, since TJFast$^+$ runs the same procedure as TJFast, there is no difference for the number of intermediate path solutions between two algorithms.

5.3 Analysis of TJFast+

In the section, we show the correctness of TJFast$^+$ and analysis its efficiency.

Theorem 5.1. *Given a twig query q and an XML database D, Algorithm* TJFast$^+$ *correctly returns all answers for q on D.*

Proof. There are two cases. The first case is that when query contains only parent-child edges, we use Algorithm 4 to evaluate it. Here the key difference between TJFast$^+$ and TJFast is in line 12,13 of *getNext*, where TJFast$^+$ pushes e_{min} to set S_n only if $e_{min} = e_{max}$. We now show the correctness of this step. Based on *tag+level* streaming, elements with the same tag name but different level numbers have been separated to different streams. Thus, it is impossible that there exists an element e' such that e' is an ancestor of e_{max} and e' involves in query answers. The second case is that when query contains both parent-child and ancestor-descendant edges, we reuse TJFast by simulating multiple streams with the same tag name to one sorted stream. The correctness is obvious if the original algorithm TJFast is correct. □

While the correctness holds for any kind of query, the I/O optimality of TJFast$^+$ holds for two cases (i) only parent-child relationships in all edges; and (ii) only ancestor-descendant relationships in *branching* edges. That is, TJFast$^+$ broadens the optimal class of TJFast by including queries with only parent-child edges.

Theorem 5.2. *Consider an XML database D and a twig query q with (i) only parent-child relationships in all edges or (ii) only ancestor-descendant edges in branching edges. 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 of this algorithm is that the number of leaf nodes in q times the length of the longest label in the input list.

It is worth to note two advantages of TJFast$^+$ over TJFast: (1) TJFast$^+$ typically scan less elements than TJFast by using stream pruning; and (2) TJFast$^+$ show even a larger query class to guarantee the I/O optimality than TJFast.

5.4 Suboptimal of TJFast+

Algorithm TJFast$^+$ still cannot guarantee to be I/O optimal for some queries. For example, consider the query $Q2$ and document $Doc2$ in Figure 9. Observe that based on *tag+level* streaming, there are still two input streams $(e, 4)$ and $(f, 4)$ in the document. In this case, we cannot say e_1 or f_1 involves in a solution without advancing the other stream, and we cannot advance any stream before knowing if it participates in a solution. Thus, optimality can no longer be guaranteed.

6 Experimental evaluation

This section presents experimental results on the performance of the algorithms we proposed, in comparison with the existing state-of-the-art algorithms.

6.1 Experimental setup

6.1.1 Setup and Data set

We implemented three XML join algorithms: TwigStack, TJFast, TJFast$^+$, in JDK 1.4 using the file system as a simple storage engine. 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 the following synthetic and real-world data sets for our experiments:

- Synthetic data: We use synthetic data from the DTD shown in Figure 3. to control the structure characteristics of the XML data. The generated XML roughly takes 40MB and contains about 2 million element nodes. Since all holistic twig join techniques focus on the core operations in XML query languages, we do not generate text values for the synthetic document[2]. The resulting data includes 2000 *books* and the depth of each subtree rooted with *book* varied from 5-20.

[2]But it does not mean that holistic twig join algorithm cannot handle text values.

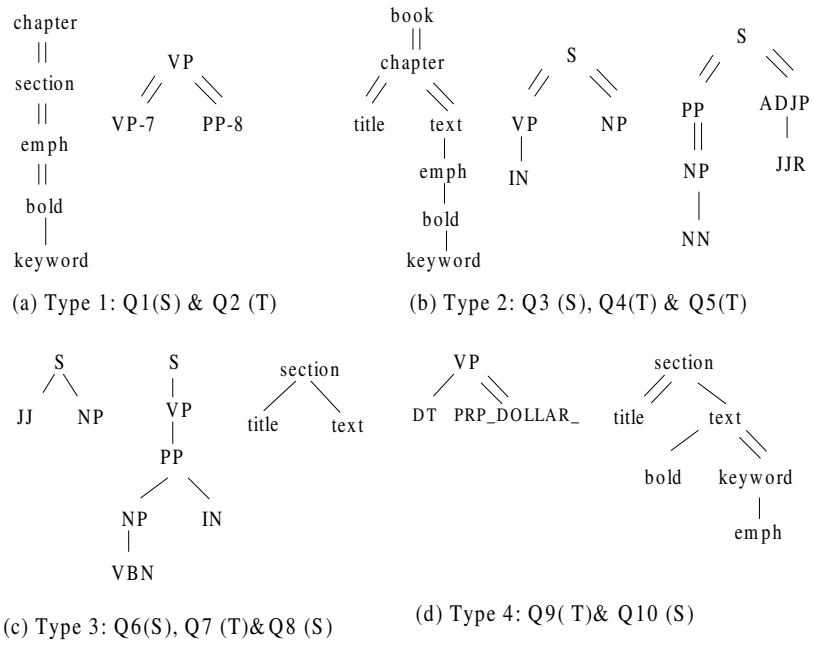


Figure 10: Ten test queries (S:Synthetic data, T: Treebank data)

- TreeBank: The real data set is TreeBank, which is downloaded from the University of Washington XML repository [16]. TreeBank comprises English sentences, tagged with parts of speech. The text nodes have been encrypted because of the copyright. Nevertheless, the deep recursive structure of this data makes it an interesting case for our experiments. TreeBank data has the maximal depth 36 and more than 2.4 million nodes.

6.1.2 Test queries

For each data set, we tested several types of XML queries (see Figure 10). The first type includes Q1 and Q2, where Q1 is a simple path queries without branching and Q2 is a twig query with only ancestor-descendant relationships. The second type includes Q3-Q5, in which all branching edges are ancestor-descendant relationships. The difference for Q2 and queries in the second type is that Q2 do not allow any existence of parent-child edges, but the second type allows the parent-child edges to appear in *non-branching* edges. The third type includes Q6-Q8 in which all edges are parent-child relationships. Finally, the fourth includes Q9,Q10 in which both parent-child and ancestor-descendant relationships appear in branching edges.

We choose these types of queries for the following reasons. All three algorithms used in

our experiments can guarantee the I/O optimality to answer queries in the first type. But only TJFast and TJFast$^+$ can provide the I/O optimality for queries in the second type. Further, only TJFast$^+$ is I/O optimal for the third type. Finally, none of algorithms are I/O optimal for queries in the fourth.

6.1.3 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 the variable length labels. 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. We used UTF-8 encoding as an efficient way to present each label, which is proposed by Tatarinov et al. [20]. In UTF-8, a variable number of bytes are used to encode different integer values. Smaller values use a smaller number of bytes. For example, if the value is smaller than 1,111,111 (by decimal 2^7=128), it is encoded with a single byte 0xxxxxxx where x represents a bit used for value encoding. The value between 1,111,111 (2^7) and 11,111,111,111 (2^{11}) are encoded with two bytes 110xxxxx 10xxxxxx , and so on. To represent an entire label with UTF-8, each component of the label is encoded in UTF-8 and then concatenated. This enables each label to be stored and compared as a variable length label, without incurring a large space overhead. 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.

It is worth noticing that there are other compression approaches which can be adopted to efficiently implement *extended Dewey* labeling scheme. For example, the recent work by O'Neil et al. [17] propose to use *bitstring* encoding to efficiently store *Dewey* labels.

6.2 Evaluation Metrics

We will use the following metrics to compare the performance of different algorithms tested in our experiments.

Number of intermediate path solutions This metric measures the total number of intermediate path solutions, which reflects the ability of algorithms to control the size of intermediate results.

Number of elements scanned and the size of disk file The former metric indicates the number of elements scanned to answer a query twig pattern. Since the label size

of an element in two labeling schemes(i.e. *region encoding* and *extended Dewey*) is different, the latter metric accurately reflects the I/O cost of algorithms of reading the input data.

Total running time This metric is obtained by averaging the total time elapsed to answer a query with six consecutive runs and the best and worst performance results discarded.

6.3 Performance Analysis

We now present experimental results comparing TwigStack, TJFast and TJFast$^+$ for a variety of scenarios. Figure 11 summarizes the performance results in total time elapsed for all queries. Table 1 shows the number of partial path solutions for three algorithms and Figure 12(a-b) shows the number of elements scanned and the size of the corresponding disk file. We begin our analysis from the results of queries Q1 and Q2.

Table 1: Number of intermediate path solutions

Type	Query	TwigStack	TJFast	TJFast$^+$	Useful
1	Q1	1748	1748	1748	1748
	Q2	2	2	2	2
2	Q3	23214	21177	21177	21177
	Q4	27389	1236	1236	1236
	Q5	1364	38	38	38
3	Q6	70988	30	10	10
	Q7	3	3	2	2
	Q8	49906	49906	46773	46773
4	Q9	10663	9	9	5
	Q10	81682	9	9	7

6.3.1 Queries of Type 1

In this section, we analyze the performance of algorithms against queries Q1,Q2. In particular, Q1 is a simple path queries without branching and Q2 is a twig query with only ancestor-descendant relationships. In [3] and Section 4,5, it is shown that algorithms TwigStack, TJFast and TJFast$^+$ are all I/O optimal for these two queries. The optimality can be validated from the observation that the number of intermediate path solutions

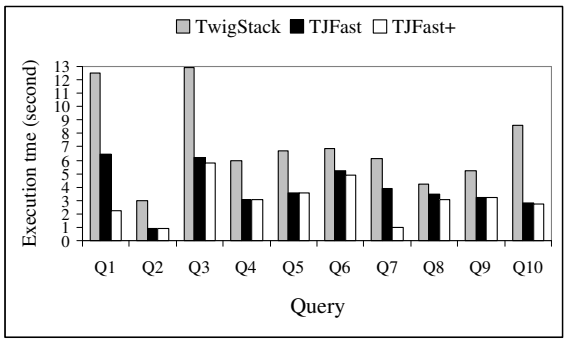


Figure 11: Total execution time

output by three algorithms equals the number of final useful path solutions (see Table 1).

From Figure 12(a-b), we see that TJFast read much less number of elements than TwigStack, and TJFast$^+$ read the least elements among three algorithms. For example, in Q1, TwigStack read 1876044 elements, but TJFast and TJFast+ only read 625772 and 26310 elements respectively. TJFast$^+$ scan only 1/70 and 1/25 of the total elements by TwigStack and TJFast. This huge gap results from the fact that TwigStack scans the elements for all nodes in the query pattern, but TSFast only scans the elements for leaf nodes. Further, TJFast$^+$ can read fewer elements than TJFast because TJFast$^+$ uses level information to prune streams. As expected, from Figure 11., we see that both TJFast and TJFast$^+$ offer better query performance than TwigStack, which suffers from the large amount of elements read and processed.

6.3.2 Queries of Type 2

In this section, we analyze the performance of algorithms against queries Q3-Q5, which include only ancestor-descendant relationships in branching edges. As shown in Table 1, TwigStack output much more intermediate path solutions than TJFast/ TJFast$^+$. For example, in Q4 and Q5, TwigStack produced 27389 and 1364 intermediate paths, while TJFast/TJFast$^+$ produced 1236 and 38 intermediate paths. More than 95% "*useless*" intermediate solutions of TwigStack are pruned by TJFast/TJFast$^+$. Furthermore, notice that the number of final solutions of Q4 and Q5 is also 1236 and 38. Thus, TJFast and TJFast$^+$ are I/O optimal, but clearly TwigStack has not this property.

As shown in Fig 11, both TJFast and TJFast$^+$ are more efficient than TwigStack for Q3-Q5. The performance enhancement is achieved by the reductions of the number of intermediate path solutions and the number of elements scanned.

6.3.3 Queries of Type 3

In this section, we analyze the performance of algorithms against queries Q6-Q8, which contain only parent-child relationships. As seen by the results in Figure 11, TJFast has better performance than TwigStack, and TJFast$^+$ has the best performance. These can be explained by two reasons: (1) TJFast$^+$ is able to scan fewer elements than TJFast, and TJFast scans fewer elements than TwigStack. For example, in Q7, TJFast$^+$ access only 736 elements, but TwigStack and TJFast need to access 551553 and 115864 elements; and (2) TJFast$^+$ is able to output less intermediate path solutions than TwigStack and TJFast. For example, in Q6, TwigStack outputs 70988 intermediate solutions, but TJFast outputs 30 and TJFast$^+$ outputs only 10. Notice that, for queries Q6, the number of final uesful path solutions is also 10. This observation confirms that TJFast$^+$ is an I/O optimal algorithm for queries with only parent-child edges, but TJFast and TwigStack are sub-optimal.

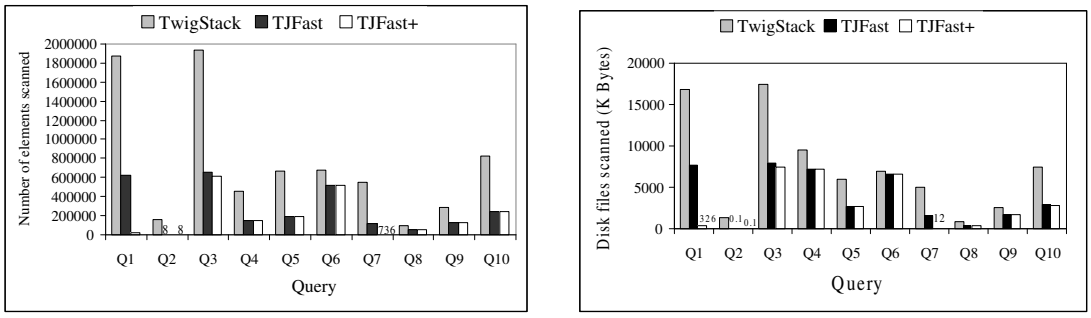
6.3.4 Queries of Type 4

Finally, in order to study the performance of three algorithms in the case of sub-optimality, we choose queries Q9,Q10, in which *branching* edges contain both parent-child and ancestor-descendant relationships. From Table 1, we can draw two conclusions. The first one is that none of algorithms can sure the I/O optimality for two queries. For example, in Q10. TwigStack produce 81682 partial path solutions, while TJFast/TJFast$^+$ produce 9 solutions. Considering the number of final path solutions is 7, all three algorithms output some redundant solutions. The second conclusion is that although no algorithms are I/O optimal in this case, TwigStack output much more "*useless*" solutions than TJFast/TJFast$^+$. Consider Q10 again. TwigStack outputs up to 9000 times *useless* solutions as many as TJFast/TJFast$^+$. As expected, TwigStack wastes too much time on outputting and processing "*useless*" partial solutions, which significantly deteriorate its performance. Therefore, as shown in Figure 11, even in the case that all three algorithms are not optimal, TJFast/TJFast$^+$ is considerably more efficient than TwigStack.

6.3.5 Summary

According to the experimental results, we draw the following three conclusions.

1. TJFast is a more efficient holistic twig join algorithm than TwigStack because it needs less disk access and identify a wider class of optimal queries. Even in the case that both TJFast and TwigStack are not optimal(see Type 4),TJFast is still much efficient than



(a)Number of elements scanned (b) Input file size

Figure 12: TwigStack, TJFast, TJFast+ for ten queries

TwigStack by significantly reducing the number of redundant intermediate path solutions.

2. Partitioning data by *tag+level* is a new source of speedup for holistic twig join, since it can further decrease the number of elements scanned and the size of intermediate results, especially when the query twig contains only *parent-child* relationships.

3. If we denote the number of elements scanned by algorithm S as $|S|$ and the number of intermediate path solution output by S as $\|S\|$, then the following two inequalities always hold in our experimental results:

$|TwigStack| > |TJFast| \geqslant |TJFast^{+}|$

$\|TwigStack\| \geqslant \|TJFast\| \geqslant \|TJFast^{+}\|$

7 Related work

With the increasing popularity of XML query processing, twig query matching is identified as a core operation in querying tree-structured XML data, there is a rich set of literatures on matching twig queries efficiently.

For binary structural join, Zhang et al.[22] proposed a multi-predicate merge join (MP-MGJN) algorithm based on (*DocId*, *Start*, *End*, *Level*) labeling of XML elements. The later work by Al-Khalifa et al [1] gives a stack-based binary structural join algorithm. Then Wu et al [21] studied the problem of binary join order selection for complex queries on a cost model which takes into consideration factors such as selectivity and intermediate results size. While many optimization strategies and indices can be applied, it is inevitable that binary structural join approach cannot guarantee each intermediate result to contribute to final answers.

Different from binary structural join approaches, Bruno et al.[3] propose a holistic twig

join algorithm, called TwigStack, to avoid producing a large intermediate result. An important idea behind TwigStack is that when we match a single path pattern of twig queries, *the nodes in other path patterns also should be considered*. Thus, providing a certain class of queries, we can ensure that each intermediate solution to a root-leaf path pattern is guaranteed to contribute to at least one final solution to the whole twig pattern.

However, the class of optimal queries in TwigStack is very small. When a twig query contains any parent-child edge, the size of intermediate results may be still out of control. For example, in our experiments, for query Q9 which contains only one parent-child relationship, the number of intermediate path solutions output by TwigStack is 10663, but finally, only 5 solutions contribute to final answers. Thus, more than 99.9% intermediate solutions are "*useless*". Thus, it is a big challenge to design a new holistic twig join algorithm to answer twig query with parent-child edges. The recent work by Jiang et al [10] studied the problem of holistic twig joins on all/partly indexed XML documents. They leverage special index structures (i.e. XR tree) on the data and achieve sub-linear performance for selective queries. Although their algorithm can skip some elements that do not participate in join, they cannot reduce the size of redundant intermediate results. In contrast, our algorithm TJFast/TJFast$^+$ answer this challenge and significantly widen the class of optimal query.

Chen et al.[4] propose BLAS, a Bi-Labeling based System, for efficiently processing complex XPath queries. They use two labeling systems, P-label and D-label to answer a query. They also propose three query translation algorithms: *split*, *push-up* and $unfold$, that translate a complex query into some basic queries. Similar to our TJFast algorithm, BLAS only needs to access labels of leaf node to process queries involving only consecutive child axes. But, in order to answer a complex query with both descendant and child axes, they have to access labels of some *non-leaf* nodes. In contrast, for any complex twig query, TJFast accesses only labels of *leaf* nodes. Furthermore, strictly speaking, BLAS is not a *holistic* twig join approach, because BLAS decomposes a complex query into several sub-queries. And when one sub-query is processed, the nodes in other sub-queries are not taken into account. Thus, BLAS faces the same fundamental problem as the techniques based on binary structural joins: many intermediate results may not be part of any final answer.

There has been much research on constructing efficient structure indexes for matching path or twig queries based on *graph* structural model. Dataguide [6] and 1-Index [15] can be used to answer simple path queries without branches. F&B index [11] can be used to answer all branching path expressions, but the size of F&B index is usually as large as the original document. So it makes this index structure not be feasible in practice. Then

Kaushik et al[13] propose A(k) index that is based on the notion of local similarity to provide a trade-off between index size and query answering power. Recently, He et al.[7] propose two workload-aware indexes: M(k) and M*(k) , which allows different index nodes to have different local similarity requirement, providing finer partitioning only for parts of the data graph targeted by longer path expression.

Kaushik et al.[12] propose to process XML Path query by using the integration of structure indexes and inverted lists. They augment the inverted lists with an *indexid*. Before structural join, they first use structural indexes to prune some *index ids*. In ordered to skip parts of the lists, they add a pointer for each entry to point to the next entry with the same indexid, called as *extent chaining*. However, the problem of *extent chaining* is that it may use more I/O cost than a linear scan when the number of lists matching an extent is high. They also propose *hybrid scan*. But the worst case for *hybrid scan* is that the entries in a list matching the selected extent are spread uniformly apart. Here we recommend them to adopt *simulation* technique proposed in Section 5.2.2 of this paper to elegantly solve this problem. In particular, each entry in the list is sorted by (*indexid*, *start*) so that all entries with the same *indexid* are clustered together. To evaluate a query, we first obtain the first entries of all desired *indexid*. We maintain a directory for this purpose. Then we simulate the corresponding multiple lists to one sorted list so that we may directly use the existing structural join algorithms. Unlike *extent chaining* or *hybrid scan*, our approach is always efficient regardless of the number of list matching an extent and the distribution of selected extents.

8 Conclusion

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 based on a novel labeling scheme: *extended Dewey ID*. Through this, not only do we reduce the disk access by only reading the labels of leaf nodes in query pattern, but we also enlarge the class of query with the I/O optimality. In addition to proposing TJFast, our contributions include: the presentation of a new data grouping strategy: *tag+level*, the development of an efficient algorithm TJFast$^+$, and a careful experimental evaluation that shows the superior performance of our algorithms over previous approaches.

There are several avenues for future work. We can extend our algorithm to efficiently process twig patterns with "NOT" predicates. For example, given an XML twig query:

$S/NOTNN[/VP][//NP]$, which finds elements that (1) are not tagged with NN, (2) have the parent element with tag S, (3) have a child element with tag VP, and (4) have a descendant element with tag NP. How can we compute answers to twig query with such predicate? Another issue is that based on *extended Dewey* labels, we may use indexes (such as B^+ index) to further improve the performance of algorithm TJFast and TJFast$^+$.

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, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal xml pattern matching. Technical report, Columbia University, 2002.

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

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

[5] P. F. Dietz. Maintaining order in a linked list. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 122–127, 1982.

[6] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. of VLDB*, pages 436–445, 1997.

[7] H. He and J. Yang. Multiresolution indexing of XML for frequent queries. In *Proc. of ICDE*, pages 683–694, 2004.

[8] H. V. Jagadish and S. AL-Khalifa. Timber: A native XML database. Technical report, University of Michigan, 2002.

[9] 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.

[10] H. Jiang, W. Wang, H. Lu, and J. Yu. Holistic twig joins on indexed XML documents. In *Proc. of VLDB*, pages 273–284, 2003.

[11] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proc. of SIGMOD*, pages 133–144, 2002.

[12] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *Proc. of SIGMOD*, pages 779–790, 2004.

[13] R. Kaushik, P. Sheony, P. Bohannon, and E. Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *Proc. of ICDE*, pages 129–140, 2002.

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

[15] T. Milo and D. Suciu. Index structures for path expressions. In *Proc. of the 1999 Intl. Conf. on Database Theory*, pages 277–295, 1999.

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

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

[18] A. Silberstein, H. He, K. Yi, and J. Yang. Boxes: Efficient maintenance of order-based labeling for dynamic XML data. In *Proc. of ICDE*, 2005.

[19] G. Stephen. *String searching algorithms*. World Scientific Publishing, 1994.

[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] Y. Wu, J. M. Patel, and H. Jagadish. Structural join order selection for XML query optimization. In *ICDE Conference*, pages 443–454, 2003.

[22] 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.