Data & Knowledge Engineering xxx (2010) xxx–xxx

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

Indexing and querying XML using extended Dewey labeling scheme

Jiaheng Lu [a,*], Xiaofeng Meng [b], Tok Wang Ling [c]

[a] The Key Lab of Data Engineering and Knowledge Engineering, MOE, Renmin University of China, China
[b] School of Information, Renmin University of China, China
[c] School of Computing, Renmin University of China, China

ARTICLE INFO

Article history:
Received 18 May 2009
Received in revised form 1 August 2010
Accepted 3 August 2010
Available online xxxx

Keywords:
XML database
XML query processing
Performance

ABSTRACT

Finding all the occurrences of a tree pattern in an XML database is a core operation for efficient evaluation of XML queries. The *Dewey* labeling scheme is commonly used to label an XML document to facilitate XML query processing by recording information on the path of an element. In order to improve the efficiency of XML tree pattern matching, we introduce a novel labeling scheme, called *extended Dewey*, which effectively extends the existing Dewey labeling scheme to combine the types and identifiers of elements in a label, and to avoid the scan of labels for internal query nodes to accelerate query processing (in I/O cost). Based on *extended Dewey*, we propose a series of holistic XML tree pattern matching algorithms. We first present TJFast to answer an XML twig pattern query. To efficiently answer a generalized XML tree pattern, we then propose GTJFast, an optimization that exploits the non-output nodes. In addition, we propose TJFastTL and GTJFastTL based on the *tag + level* data partition scheme to further reduce I/O costs by level pruning. Finally, we report our comprehensive experimental results to show that our set of XML tree pattern matching algorithms are superior to existing 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 [3,30], XQuery [4]) are typically trees, *tree pattern matching* is the central issue. For example, the following query

“`Q=//book[ author=“Chen”] //chapter/title`”

can be represented as a twig (small tree) pattern. Intuitively, it returns the `title` of `chapter` for a `book` that has an author named by “`Chen`”.

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 [1,7,12,14,16,17,19,31,43] 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 tree 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 pre-order [15,24]) or textual positions of *start* and *end* tags (e.g. region encoding [5]) or path expressions(e.g. Dewey ID [20,21,37]) or prime numbers (e.g. [44]). By applying these labeling schemes, one can determine the relationship (e.g.

* Corresponding author. School of Information, Renmin University of China, Beijing 100872, China.
E-mail address: jiahenglu@gmail.com (J. Lu).

0169-023X/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2010.08.001

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*, and *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 other useful information is provided*.

In this article, motivated by the existing *Dewey ID* [37], 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, Fig. 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 an XML tree pattern.

To perform structural joins efficiently, several algorithms have been developed to process twig queries. In particular, Bruno et al. [5] proposed the holistic twig matching algorithms PathStack/TwigStack. For evaluating queries with only *ancestor–descendant* edges, TwigStack guarantees that each intermediate path solution contributes to final answers. However, when queries contain any *parent–child* relationship, TwigStack is non-optimal since it may output a large size of intermediate matches to the individual path expressions which do not contribute to final answers. Algorithm TwigStackList [26] guarantees the optimality for queries in which *parent–child* relationships only occur under the *non-branching* query nodes and thus slightly enlarge the optimal query class. In addition, the recent algorithms *Twig*^{2}*Stack* [8] and TwigList [33] share the same procedure with TwigStack and TwigStackList: they label document with region encoding scheme and add any element in the corresponding structure if it satisfies the sub-query rooted at the matching query node. Although there are a rich set of literatures on efficient XML tree pattern matching, unfortunately, most of them only consider region encoding labeling scheme and they do not exploit the opportunity to develop more powerful labeling schemes to speed up the efficiency of XML tree pattern matching.

In this article, we present four tree pattern matching algorithms based *extended Dewey* labeling scheme. Fig. 2 summarizes the four algorithms and their properties.

- TJFast To match a twig pattern, TJFast 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.
- GTJFast Since XQuery may contain multiple path expressions in the FOR, LET,WHERE and RETURN clauses, all with different semantics, XQuery statements cannot be modeled as simple tree patterns. We treat XQuery statements as generalized tree patterns (GTP [5,9]). In order to efficiently answer GTP, we propose GTJFast to efficiently exploit the non-output nodes to reduce the I/O cost.
- TJFastTL and GTJFastTL We propose TJFastTL and GTJFastTL by extending TJFast and GTJFast based on tag + level data partitioning ([11]). In the tag + level partition scheme, two elements appear in the same list if and only if they have the same tag and level number. 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. In addition, we prove that TJFastTL and GTJFastTL algorithms guarantee the optimality for queries with only parent–child relationships.

We implemented eight XML tree pattern matching algorithms: TJFast,TJFastTL, GTJFast, GTJFastTL, TwigStack [5], TwigStackList [26], iTwigJoin [10] and *Twig*^{2}*Stack* [8], where TJFast, TJFastTL, GTJFast and GTJFastTL are novel algorithms proposed in this article,

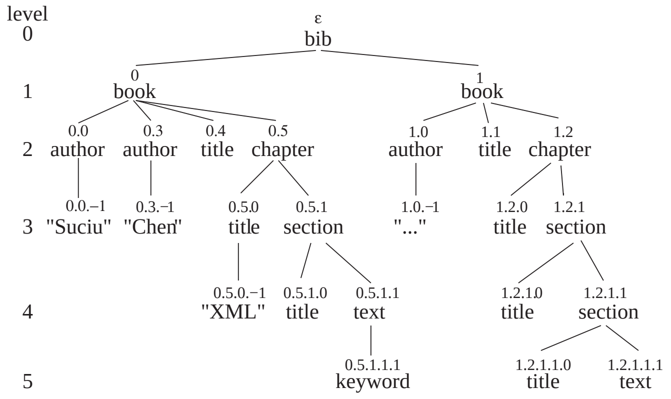
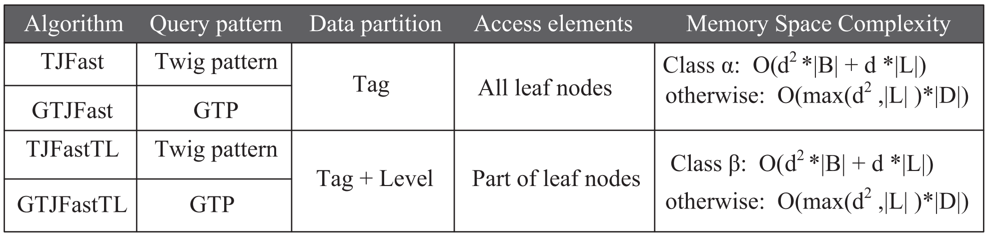


Fig. 1. An XML tree with *extended Dewey* labels.



Algorithm	Query pattern	Data partition	Access elements	Memory Space Complexity
TJFast	Twig pattern	Tag	All leaf nodes	Class α: $O(d^2 * \|B\| + d * \|L\|)$ otherwise: $O(\max(d^2, \|L\|) * \|D\|)$
GTJFast	GTP			
TJFastTL	Twig pattern	Tag + Level	Part of leaf nodes	Class β: $O(d^2 * \|B\| + d * \|L\|)$ otherwise: $O(\max(d^2, \|L\|) * \|D\|)$
GTJFastTL	GTP			

Fig. 2. Properties of algorithms proposed in this article (Class α: queries with only A–D relationship in branching nodes; class β: queries with only A–D relationships in branching nodes or queries only P–C relationships; d: the length of the longest path in the document; $|B|$: the number of branching nodes; $|L|$: the number of leaf nodes).

and the other four are existing approaches. Experimental results on a variety of queries and data sets demonstrate the significant superiority of our algorithms over the previous algorithms in terms of *the number of elements scanned*, *the size of intermediate results* and *query performance*.

1.1. Organization

The rest of this article is organized as follows. Some preliminaries, including a brief description of *Dewey ID* labeling scheme are covered in Section 2. In Section 3, we give introduce the new labeling scheme, named *extended Dewey*. Section 4 presents a holistic twig pattern matching algorithm TJFast and Section 5 extends it to process a generalized tree pattern. Section 6 develops algorithms TJFastTL and GTJFastTL based on *tag* + *level*. Section 7 presents thorough experimental studies about the performance comparison between the novel algorithms and the prior methods. Finally, Section 8 shows the related work and Section 9 concludes the paper.

2. Preliminaries

2.1. Data model and XML twig pattern

We model XML documents as *rooted*, *ordered* and *tagged* 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 Q with n nodes can be represented as a list of *n-ary* tuples, where each tuple $(q_1, \cdots, q_n)$ consists of the database elements that identify a distinct match of Q in D.

2.2. Dewey ID labeling scheme

Tatarinov et al. [37] 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 7, this method faces the problem of a 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.

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 the set of all (distinct) names of children of t. This clue is easily derived from DTD, XML schema or other schema constraints. For example, consider the DTD in Fig. 3; 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, because we can scan the document once to get the necessary *child names clue* before labeling the XML document. In addition, in the case that specifications like “ANY” in DTD or “xsd:any” in XML Schema are used to allow appearance of any tag names defined the DTD or XML Schema, we cannot only read DTD or XML schema to understand the *child names clue*. The whole XML document has to be scanned to find the concert element types hidden in “ANY” specification.

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. Intuitively, an ideal labeling scheme (i) should record the order of sibling elements, and (ii) we can derive the element name from its label according to $CT(t)$. But the existing Dewey labeling scheme cannot fulfill both requirements. Motivated by this, we next show a new labeling method to use modulo function to enable name-awareness with order maintainability. More precisely, 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 that, 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 $CT(t_s)$ ($k\in\{0,1,...,n-1\}$), 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

$$x=\begin{cases}\left\lfloor \frac{y}{n}\right\rfloor\cdot n+k & \text{if } (y \bmod n)<k;\\ \left\lfloor \frac{y}{n}\right\rfloor\cdot n+k & \text{otherwise.}\end{cases} \quad \text{where } n \text{ denotes the size of } CT(t_s).$$

Example 3.1. Fig. 1 shows an XML document tree that conforms to the DTD in Fig. 3. We assign the labels according to the pre-order traversal of the tree. 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

```
<!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) *>
```

Fig. 3. DTD for XML document in Fig 1.

$x = 4 / 3$ $3 + 2 = 5$. So chapter is assigned the label "0.5". Note that extended Dewey labels maintain the order of children under the book. Our function for label generation explicitly considers the order of sibling nodes.

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

Theorem 3.2. *The extended Dewey does not alter the asymptotic space complexity of the original Dewey labeling scheme.*

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 3.3. (Function $F(t,x)$) Let Z denotes the non-negative integer set and Σ denotes the alphabet of all distinct tags 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 $k = x \bmod n$.

Definition 3.4. (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, and o), where (i) the input set $I = Z \cup \{-1\}$; (ii) the set of states $S = \Sigma \cup \{\text{PCDATA}\}$, 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 3.5. Fig. 4 shows the FST for DTD in Fig 3. 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".

It is worth noting three points about FST: (i) the memory size of the above FST is quadratic in the number of distinct tags in XML documents, as the number of transition 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. Dynamic XML labeling

We next discuss how to support updates in XML documents based on extended Dewey labeling scheme. One way is to use ORDPATH [32], where odd numbers are used to represent the order coding and even numbers are reserved for delimiters. Assuming a DTD is $a \rightarrow (b|c)$, for instance, considering an XML document with the root a_1 that has three children b_1, c_1 and c_2, their respective extended Dewey labels are assigned 1, 2 and 4 (a_1 is labeled ε), and dynamic extended Dewey labels should be assigned 1, 3 and 7 instead. If a new element b_2 is inserted between c_1 and c_2, then it could be elegantly assigned 3.4.1 without updating any

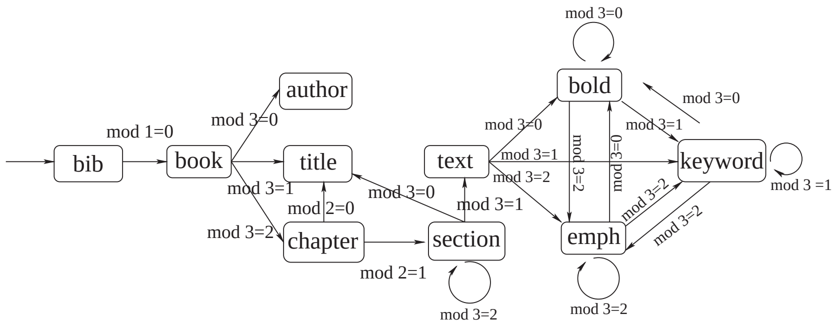


Fig. 4. A sample FST for DTD in Fig 3.

existing label. The value 4 (or any even value in any nonterminal component) represents a caret only, that is, it doesn't count as a component that increases the depth of the node in the tree.

More precisely, we use *label*(u) to denote the *dynamic extended Dewey* label of element u. For each u, *label*(u) is defined as *label*(s).x, where s is the parent of u. For any element u with parent s in an XML tree, "(1)" if u is a text value, then $x = \lambda$ (we do not use -1 as it may be used for internal nodes); "(2)" otherwise, assume that the element name of u is the *k-th* tag in $CT(t_s)$ ($k \in \{0,1,...,n-1\}$), where t_s denotes the tag of element s.

(2.1.) if u is the first child of s, then $x = 2k + 1$;
(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

$$x = \begin{cases} \left\lfloor \frac{y}{n} \right\rfloor \cdot n + 2k + 1 & \text{if } (y \bmod n) < k; \\ \left\lfloor \frac{y}{n} \right\rfloor \cdot n + 2k + 1 & \text{otherwise.} \end{cases} \quad \text{where } n \text{ denotes the size of } CT(t_s).$$

In order to insert a new node n between any two siblings of a parent node X (known as creating in), we can create a component with an even ordinal falling between the final (odd) ordinals of the two siblings, then following this with a new odd component, usually 1. To insert new nodes to the left of all existing children of any node, we can use the nearest smaller even ordinal than the first odd component. In interpreting a dynamic extended Dewey label, the even components (carets) simply do not count for ancestry: 3.5.6.2.1 is a child of 3.5, and a grandchild of 3.

To derive the path sequence from dynamic extended Dewey labels, all even components are not counted. Given a tag t in T, suppose $CT(t) = \{t_0, t_1, \cdots, t_{n-1}\}$, a dynamic function $DF(t,x)$: $\Sigma \times Z \rightarrow \Sigma$ can be defined by $DF(t,x) = t_{(k+1)/2}$, where $k = x \bmod 2n$. Given a *dynamic extended Dewey* label, we can use a dynamic FST to translate the label into a sequence of element names. Let I denote the integer set. DFST is a 5-tuple (N, S, i, δ, and o), where (i) the input set $N = I \cup \{\lambda\}$; (ii) the set of states $S = \Sigma \cup \{\text{PCDATA}\}$, 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 $x = \lambda$, $\delta(t,x) = PCDATA$, otherwise $\delta(t,x) = DF(t,x)$. No other transition is accepted. (v) The output value o is the current state name. Follow-up of the above example, assuming a DTD is $a \rightarrow (b|c)^*$ and the root is a, given a label 3, $k = 3 \bmod (2*2) = 3$, then $t_{(3+1)/2} = t_2 = c$. Therefore the path is a/b.

The fact that insertions require no relabelings of old nodes is extremely important to insert performance and possible concurrency of operations [15,32]. With dynamic extended Dewey labels, all path information can be derived from labels and all existing labels remain the same with any insertion of sequence.

4. Twig pattern matching

4.1. Path matching algorithm

It is quite straightforward to evaluate a query path pattern in our approach. Note that 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. Much research has been done on this topic and there are a rich set of algorithms on efficient string processing with *don't care* symbols, e.g. see [2,22].

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 [5]), 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, which accesses only labels of leaf query nodes. The basic idea in TJFast is to first output the root-leaf path solutions and then merge those intermediate solutions to get the final results. 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 of Fig. 5, dbl(a) = {b,c} and dbl(c) = {f,g}. In addition, *topBranchingNode* denotes the highest branching nodes in the query Q.

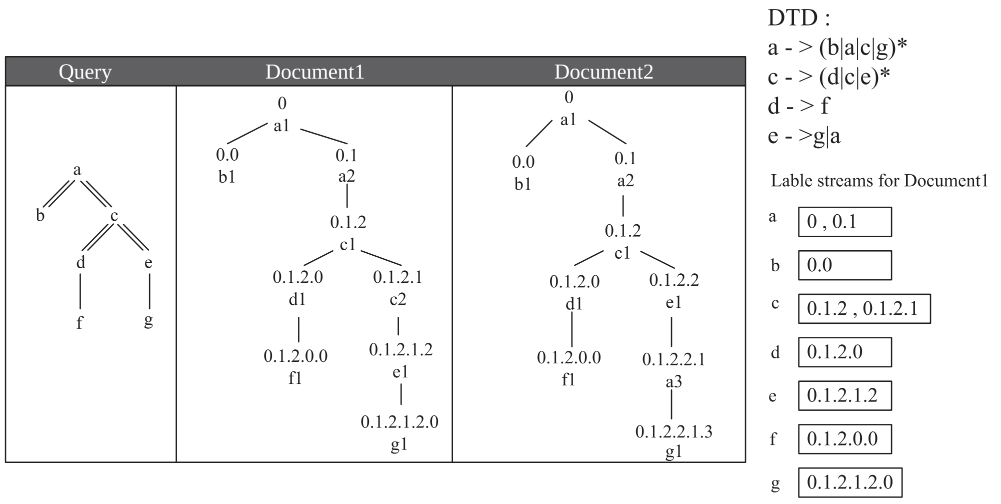


Fig. 5. Example twig query and documents.

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.

Algorithm 1 TJFast.

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

Procedure locateMatchedLabel(f)
/* Assume that the path from the root to element get(T_f) is $n_1/n_2/\cdot\cdot\cdot/n_k$ 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
   $b \rightarrow e \in S_b$)
```

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 (lines 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.

It is not difficult to understand the main procedure of TJFast (see Algorithm 5). In lines 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 the highest branching node. In line 6, we output some path matching solutions in which each element that matches 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.[1]

Algorithm 2 getNext(n).

```
1:  if (isLeaf($n$)) then return $n$
2:  else for each $n_i \in$ dbl($n$) do
3:          $f_i = getNext(n_i)$
4:          if ($isBranching(n_i) \wedge empty(S_{ni})$) then return $f_i$
5:          $e_i = max\{p|p \in MB(ni, n)\}$
6:      max = $maxarg_i\{e_i\}$
7:      min = $minarg_i\{e_i\}$
8:      for each $n_i \in$ dbl($n$) do
9:          if ($\forall e \in MB(n_i, n)$: $e \notin ancestors(e_{max})$) then return $f_i$
10:     for each $e \in$ MB($n_{min}$, $n$) do
11:         if ($e \in ancestors(e_{max})$) then updateSet($S_n$, $e$)
12:     return $f_{min}$
```

```
Function MB($n$, $b$)
1:  if (isBranching($n$)) then Let $e$ be the maximal element in set $S_n$
2:  else Let $e$ = current($T_n$)
3:  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 the set $S$ such that $a \notin$ ancestors(e) and $a \notin descendants(e)$
```

```
Procedure updateSet($S$, $e$)
1:  clearSet($S$, $e$)
2:  Add $e$ to set $S$
```

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 decide the path solutions that can be output in Procedure *outputSolutions*. In line 18 of Algorithm 2, 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.

Example 4.1. Consider the query and Doc1 in Fig 7. 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 the query). 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 $c_2 \notin S_c$.

4.3. Analysis of TJFast

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

Lemma 4.2. *In Procedure clearSet of Algorithm TJFast, any element* e *that is deleted from set* S_b *does not participate in any new solution.*

[1] 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 match the query "a/b", but this solution has been output by another element ending with b_1.

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 4.3. *In line 18 of Function getNext, if element* $e \notin$ *ancestors*(e_{max}) *and* $e \notin S_n$*, then* e *is guaranteed not to 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 4.1, each deletion operation is *safe*. Therefore, the lemma is proved. □

Lemma 4.2 shows that any element deleted from sets does not participate in new solutions, so the deletion is *safe*. Lemma 4.3 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 4.4. *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 4.5. *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* $O(d^2*|B|+d*|L|)$*, where* $|L|$ *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 $e \in$ ancestors(e_{max}), $e\in$ 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 $O(d^2)$.

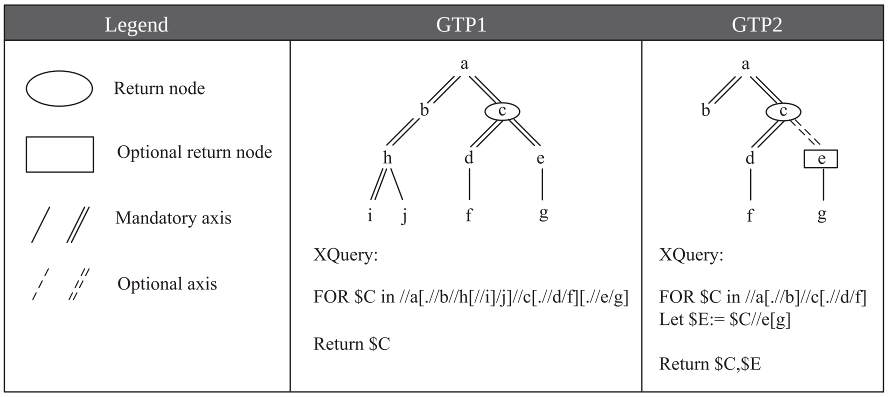
Theorem 4.5 holds only for query with *ancestor–descendant* relationships to connect *branching* nodes. Unfortunately, in the case where the query contains *parent–child* relationships between *branching* nodes and their children, Algorithm TJFast is no longer guaranteed to be I/O optimal. For example, 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.

5. Generalized tree pattern matching

5.1. Generalized tree pattern

While XQuery expression evaluation includes the matching of tree patterns, and hence can include tree pattern evaluation as a component, there is much more to XQuery than simply tree pattern [9]. In particular, the possibility of quantification in conditions (e.g., EVERY), the possibility of optional elements in a return clause, and many different forms of return results, all involve much more than merely obtaining a tree pattern evaluation. Therefore, in this section, we study generalized tree pattern (GTP) matching [9]. Intuitively, a GTP additionally defines optional axis and optional return nodes to represent more semantics than a simple tree pattern. Fig. 6 depicts two sample XQuery statements and their respective GTPs.

In GTP1, node c is a return node, i.e., only its result is of interest. In GTP2, node e is optional (in general, any expression in the LET or RETURN clauses is optional) in the sense that a c element can be a match even without any descendant e elements. Any matching e elements, however, must be grouped together under their common c ancestor elements.



XQuery:

```
FOR $C in //a[.//b//h[//i]/j]//c[.//d/f][.//e/g]

Return $C
```

XQuery:

```
FOR $C in //a[.//b]//c[.//d/f]
Let $E:= $C//e[g]

Return $C,$E
```

Fig. 6. Examples of GTP.

Existing works on holistic twig query processing focus only on returning the entire twig results [26,28,33]. In a GTP evaluation, returning the entire twig results is seldom necessary and may consequently cause duplicate elimination and/or ordering problems. Next we propose a new holistic algorithm, called GTJFast, to efficiently process GTPs without those expensive post-processing.

5.2. *GTJFast algorithm*

In the following, we first present the intuition in the optimization to process the non-return nodes and optional return nodes in GTP, and then formally show our algorithm `GTJFast` followed by theoretical analysis.

5.2.1. *Optimization on non-return nodes*

Non-return nodes do not appear in the final results, so all elements of such nodes are not required to be buffered during the processing. But we are still interested in their existences to determine the output nodes. Therefore, it is important to *judiciously* and *compactly* record the existences of non-return elements.

Intuitively, see the example in GTP1 again in Fig. 6. We classify all nodes in queries to four categories according into their properties.

(1) Non-branching and non-return nodes An example is node *b* in GTP1. For this type of nodes, we can easily derive the whole path by *extended Dewey* labeling scheme and easily determine whether they contribute to one path solution or not. Therefore, it is not necessary to buffer the elements of this type in the main memory.
(2) Non-branching and return nodes An example is node *e* in GTP2. This type of nodes should be stored in the main memory. However, as they are not branching nodes, we do not remember their children information.
(3) Branching and non-return nodes An examples is node *h* in GTP1. Those elements are not required to buffer in the main memory, but as they are branching nodes, we need to maintain the information of their descendants and children, which can achieved by "*BitArray*" in our algorithm.
(4) Branching and return nodes An examples is node *c* in GTP1. Those elements have to be buffered in the main memory and the corresponding descendants/children relationship should be recorded (by "*BitArray*").

5.2.2. *Optimization on optional return nodes*

The existence of optional return nodes in the final answer is not mandatory. Take GTP2 as an example, node *e* is an optional return node. That is, a *c* element can be a match even without any descendant *e* elements. Any matching *e* elements, however, must be grouped together under their common *c* ancestor elements. To process optional return nodes, the main approach is the same and the only additional job is to mark the optional branch, and check its existence when outputting the results.

5.2.3. *Algorithm*

Next we give the description of our algorithm, starting with the introduction of some data structures.

Data structure GTJFast keeps a set S_b for branching node *b* during execution, and each two elements in set S_b have an *ancestor–descendant* or *parent–child* relationship. Different from TJFast, each element *e* with type *b* in the set is associated with a "BitArray(e)" and "outputList(e)". The length of a BitArray equals the number of children of *b* in the query. Given a child *c* of *b*, BitArray(e,c) = 1 if the corresponding relationship (P–C or A–D relationship) between *c* and *b* is satisfied in the data. In addition, each element e' in *outputList(e)* is a potential final result related to *e*, where b' is a child/descendant node of *b* in query if the type of e' is b', as illustrated in the following example.

Example 5.1. See the query and example document in Fig. 7, the BitArray of a_1 is "11", which shows that a_1 has the corresponding two children b_1 and c_2. Since c is the return node, a_1 is associated with c_2. Similarly, the BitArray of a_2 is "11", and is associated with c_1.

Algorithm The main idea in GTJFast (see Algorithms 3 and 4) is the same as that in TJFast. That is, only the labels of leaf query nodes are accessed, and the partial matching results are maintained in sets, and finally the results are output and merged. However, the differences between GTJFast and TJFast are summarized as follows.

(1) In the procedure *updateSet*(*S*,*e*), GTJFast additionally bottom-up update the BitArray's of *e* and its ancestor branching nodes as well.
(2) In the new procedure *emptyAllSets*(q) which does not exist in TJFast, GTJFast clears the set S_q and recursively clears all sets $S_{q'}$, $q' \in$ children(q). For each element *e* to delete in S_q, in the case where all bits in BitArray(*e*) are true, showing that the subtree rooted with *q* is satisfied. If *q* is an output nodes, GTJFast adds *e* to the corresponding *outputList*, and if *q* is the top branching nodes, GTJFast outputs elements in the corresponding *outputList*.
(3) In the procedure *mergeAllPathSolutions*(), unlike TJFast where the output elements may be useless to the final results, all elements output in GTJFast are guaranteed to contribute to the final results. The merge algorithm is implemented by the traditional sorted multiple-way merge join.

Algorithm 3 GTJFast.

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

```
Procedure emptyAllSets($q$)
1: if ($q$ is not a leaf node) then
2:   for each child $c$ of $q$ do
3:     emptyAllSets($c$);
4:   end for
5: end if
6: for Each element $e$ in $Sq$ do
7:   emptySet($q$, $e$);
8: end for
```

```
Procedure emptySet($q$, $e$)
1: if (all bits in BitArray($e$) are "1") then
2:   if ($q$ is an output node) then
3:     Add $e$ to the output list of the nearest ancestor branching node of $q$;
4:   end if
5:   if ($q$ is the top branching node) then
6:     output the outputList($e$);
7:   end if
8: end if
```

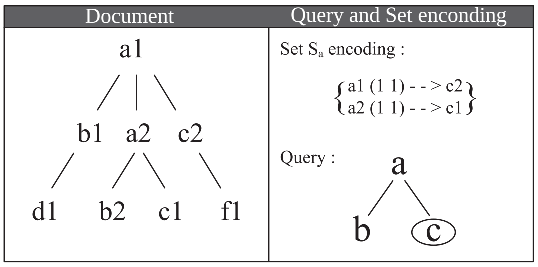


Fig. 7. Example of set encoding in GTJFast.

Example 5.2. Consider the query and document in Fig. 8. g is the only return node. A set is associated with each branching node. Table 1 shows the states of the sets when the *getNext*( ) repeatedly returns the current elements. In the first iteration, c_1,d_1,f_1 and g_1 are scanned, and c_1 is return from *getNext.* As b_1 has two children, its bitArray is “11”. After d_1 is returned, b_1 is deleted from S_b and its existence is recorded in a_1 (10), which shows that the subtree rooted with b_1 satisfy the query. Consequently, f_1, f_2, g_1 and g_2 are returned from *getNext*. When g_1 is returned, it is an output node and we add it to the result lists of e_1 and e_2. When g_1 is returned, it is transferred to S_a. As the BitArray of a_1 is “11”. g_1 is a final result. Note that we do not explicitly process g_2, as it does not satisfy the query path pattern.

Algorithm 4. updateSet(S, e).

```
Procedure updateSet($S$, $e$)
1: clearSet($S$, $e$);
2: add $e$ to set $S$;
3: Assume that the query node type of $S$ is $q$;
4: for $\forall q'$, s.t. $q'$ is a child of $q$ do
5:   if ($\exists e'$ with type $q'$ s.t. $e'$ satisfy the relationship between $q$ and $q'$)
6:     BitArray($e$, $q'$) = 1;
7:   else BitArray($e$, $q'$) = 0;
8: end for
9: if (all bits in BitArray($e$) are “1”) then
10:   if ($q$ is not the top branching node) then
11:     updateAncestorSet($q$, $e$);
12:   else
13:     output the outputList($e$);
14:   end if
15: end if
```

```
Procedure updateAncestorSet ($q$, $e$)
1: Assume the nearest ancestor branching node of $q$ is $q'$ in the query;
2: for any element $e'$ in $S_{q'}$ do
3:   if (BitArray($e'$, $q$) == “0”) then
4:     Set BitArray($q'$, $q$) be “1”;
5:     Add all elements in outputList($e$) to outputList($e'$);
6:     if ((all bits in BitArray($b$) are “1”) $\wedge$ (q is not the top branching node)) then
7:       updateAncestorSet ($q'$, $e$);
8:     end if
9:   end if
10: end for
```

5.3. Analysis of GTJFast

In this section, we first show the correctness of GTJFast and then analyze its complexity. Before proceeding, we need a preliminary lemma.

Lemma 5.3. *In Algorithm GTJFast, suppose any element e is removed from set S_q, then e matches the subtree rooted with q if and only if all bits in BitArray(e) are “1”.*

Proof. In Procedure updateSet of Algorithm GTJFast, when any new element e is inserted to S_q, BitArray(e,q') = 1, where q' is one of children of q, if and only if there is an element e' with the type q' such that e' and e satisfy the relationship between q and q'

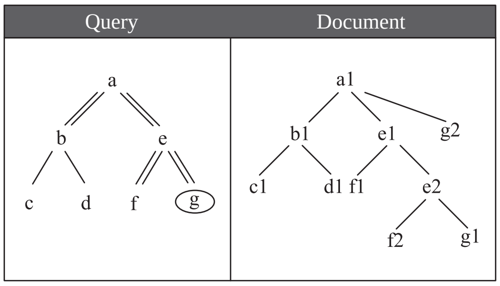


Fig. 8. An example to illustrate GTJFast.

Table 1
Set encoding (g_2 does not satisfy the query path pattern, so is excluded).

Returned node from getNext	Set(a)	Set(b)	Set(e)
c_1	$\{a_1\ (1\ 0)\}$	$\{b_1\ (1\ 1)\}$	NULL
d_1	$\{a_1\ (1\ 0)\}$	$\{b_1\ (1\ 1)\}$	NULL
f_1	$\{a_1\ (1\ 1)\}$	$\{b_1\ (1\ 1)\}$	$\{e_1\ (1\ 1)\}$
f_2	$\{a_1\ (1\ 1)\}$	$\{b_1\ (1\ 1)\}$	$\{e_2\ (1\ 1), e_1\ (1\ 1)\}$
g_1	$\{a_1\ (1\ 1)\rightarrow g_1\}$	$\{b_1\ (1\ 1)\}$	$\{e_2\ (1\ 1), e_1\ (1\ 1)\}$

(lines 4–10). Therefore, when all bits of BitArray(e) are “1”, there exist elements to satisfy all P–C and A–D relationships for element e. Furthermore, in Procedure updateAncestorSet, the matching relationships are recursively updated bottom-up to the root in lines 2 to 10. Therefore, e matches the subtree rooted with q if and only if all bits in BitArray(e) are “1”. □

Using the lemma above, we can see that whether or not an element is a query answer is exactly reflected by the values of the corresponding *BitArray*. Further, by lines 5–7, in Procedure *emptySet*, all correct solutions are output. Therefore, we have the following result.

Theorem 5.4. *Given a generalized tree pattern Q and an XML database D, Algorithm GTJFast correctly returns all the answers for Q on D.*

Proof. In Procedure clearSet of Algorithm GTJFast, any element e that is deleted from set S_b does not participate in any new solution. Function *getNext* guarantees that any element e that matches a branching node and participates in any final answer occurs in the set S_b. Thus the insertion is complete. In Algorithm GTJFast, suppose any element e is removed from set S_q, where q is the top branching node in Procedure updateSet(q), then e matches the whole query if and only if all elements in outputList(e) belong to final query answers by Lemma 5.3, as all bits in BitArray(e) are “1” in this case. So, we get the correctness of the algorithm. □

While the correctness holds for any given GTP, the I/O optimality holds for a subset of queries. In these cases, GTJFast guarantees that each output path solutions belong to final answers.

Theorem 5.5. *Consider an XML database D and a generalized tree query Q with only ancestor–descendant relationships between branching nodes and their children. The worst case I/O complexity of GTJFast is linear in the sum of the sizes of input and output lists. The worst case space complexity is* $O(d^2 * |B| + d * |L|)$, *where* $|L|$ *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. The proof is similar to that in Theorem 4.5. GTJFast makes an optimization for non-output nodes and optional axis, and its space and I/O cost is no greater than GTJFast in the worst case. □

The above theorem holds only for query with *ancestor–descendant* relationships connecting *branching* nodes. When the query contains *parent–child* relationships between *branching* nodes and their children, the worst case I/O complexity of GTJFast is still linear in the sum of the sizes of input and output lists. As for memory space complexity, however, Algorithm GTJFast may contain the whole document in the main memory and thus its space cost is $O(m * |D|)$, where $m = max(d^2, |L|)$, where d is the length of the longest label in the input lists, and $|L|$ is the number of leaf nodes in Q.

However, we claim that the worst case will not practically happen unlike the traditional main memory XML database that always stores the entire DOM tree in the memory before processing [3,8,41]. GTJFast only stores document elements that satisfy some part of GTP at runtime. More specifically, there are the following constraints on the data to be stored. First, the elements that have labels matching with the query need to be stored. Second, when the selectivity of GTP is high, only small portion of elements will be pushed in the set. Third, only elements for output nodes which satisfy the query subtree are stored. The elements which satisfy the smaller subtree, but not the whole query tree, would be removed from the main memory when during the runtime. Hence, it is unlikely to keep the entire document in the memory in practice. In addition, as shown in the next subsection, the space cost of GTJFast is often smaller than state-of-the-art algorithm *Twig²Stack* [8] which also requires to buffer the whole document in the worst case.

5.4. Comparison between GTJFast and Twig²Stack

To compare *GTJFast* with *Twig²Stack*, two algorithms may contain the whole document in the main memory in the worst case. But as illustrated in Example 6 and our experimental results, *GTJFast* takes much smaller space cost than *Twig²Stack* in most cases due to the compact *BitArray* encoding. Furthermore, *GTJFast* reads labels of only leaf query nodes and significantly reduces I/O cost. Finally, in the case when the main memory is limited and cannot load the whole document, *GTJFast* guarantees high quality of intermediate results for a large class of queries as shown in Theorem 5.5. But *Twig²Stack* fails to provide such guarantee in theory and practice with respective to the case of small memory.

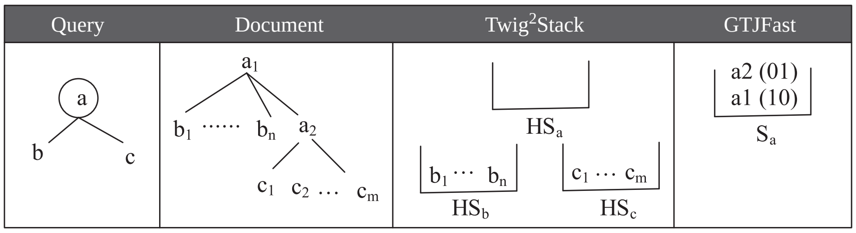


Fig. 9. Comparison *GTJFast* and *Twig*^{2}*Stack*.

Example 5.6. Consider the query and data tree in Fig. 9. The output node is node a and note there is no answer in this example. But *Twig*^{2}*Stack* needs to buffer all b_1 to b_n and c_1 to c_m in the main memory by postorder traversal. However, *GTJFast* only adds a_1 and a_2 to the set S_a and use BitArray to elegantly record the matching relationships of b and c, avoiding the storage of useless elements. Therefore, although the worst case of both *GTJFast* and *Twig*^{2}*Stack* requires to store the whole document in the main memory, *GTJFast* may store less elements than *Twig*^{2}*Stack* by the set encoding of BitArray to avoid the storage of useless nodes.

6. Tree pattern matching on *tag* + *level*

Algorithms TJFast and GTJFast need to process each node in the input lists to check whether or not it is part of an answer to the query (tree or GTP) pattern. When the input lists are very long, this may take a lot of time. In this section, we propose the use of *tag* + *level* data partition scheme on the input lists to prune data by levels and thus speed up this processing. We begin this section by introducing our motivation.

6.1. Motivation

To understand the motivation of *tag* + *level* partition, see the query and data tree in Fig. 10. There are two input streams T_c, T_d in the example XML data and their first elements are c_1 and d_1 respectively. In this case, TJFast cannot say c_1 or d_1 involves in a solution without advancing other stream. Thus, optimality can no longer be guaranteed. To overcome this sub-optimality, our idea is to separate c_1,c_2,d_1, and d_2 into four different streams so that the algorithm can "*see*" c_1,c_2,d_1, and d_2 simultaneously. That is, the current cursors point to all four elements c_1,c_2,d_1, and d_2 simultaneously. *Tag* + *level* partition is a *refinement* of data partition for the previous *tag* partition. In *tag* + *level*, two elements belong to the same stream if and only if they have the same *element names* and *level numbers*.

6.2. Level pruning

To answer a twig query, before structural join, we explore the advantage of *tag* + *level* partition and prune away those streams in which all elements do not involve in the solutions. Given the knowledge of the level number, we safely skip streams that cannot find *matching streams*. For example, consider the query and data in Fig. 10 again. Based on *tag* + *level* partition, there are five input streams: $(c,2)$,$(c,3)$,$(c,4)$, $(d,2)$,$(d,3)$, where the first field of each tuple denotes *stream tag* and the second denotes *level number*. Observe that both (b,c) and (b,d) are *parent–child* relationships. Therefore, we may safely skip all elements (i.e. c_3, c_4) in stream $(c,4)$, since there is no $(d,4)$ stream at all. Algorithm 5 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.

It turns out that when the twig query contains more *parent–child* edges, the effect of level 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, which dampens the effect of level pruning.

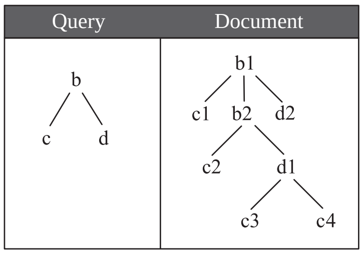


Fig. 10. Illustration to the necessary of *tag* + *level* data partition. When TJFast scan c_1 and d_1, it cannot immediately determine whether any of them contributes to final answers.

6.3. Holistic algorithms TJFastTL and GTJFastTL based on tag + level

We next show how to extend *TJFast* and *GTJFast* algorithms based on *tag* + *level* partition. In particular, when queries contain only parent–child relationships in all edges, we may develop an algorithm to guarantee the I/O cost optimality. But when queries include any ancestor–descendant edges, we cannot make such good claim. Therefore, we describe two algorithms as follows based on the query structure to achieve the respective best-effort fashion without compromising the optimality of holistic matching results.

6.3.1. Query with only parent–child edges

Before evaluating the query, we first cluster streams to several *matching groups* so that only elements in the same group potentially match the tree query. For example, consider the XML data in Fig. 10 again. There are two groups which likely match the query. One is $(c,2)$,$(d,2)$ streams, and the other is $(c,3)$, $(d,3)$. But, for example, the combination of $(c,2)$,$(d,3)$ unlikely provides any query solution.

Algorithm 5 Level 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 participate in 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 (ni, n) is a parent–child edge then
5:      delete elements in levels(n) that cannot find child level in levels(ni)
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
```

The only changes of algorithms are in line 8 and line 17 of *getNext* function. In line 8, since there is only one element to match branching node n, we do not use *max* function and directly let $e_i = MB(n_i,n)$. In line 17, $e_{min} = e_{max}$ means all e_i are the same element. Recall that Algorithm TJFast (and GTJFast) can only ensure that all e_i have *ancestor–descendant* relationships, which possibly causes its sub-optimality for some queries. But in the current scenario, since all e_i are identical, e_{min} matches branching node n in the path solutions of current element of each stream T_{f_i}. Therefore, when query contains only *parent–child* edges, *TJFastTL* (and GTJFastTL) guarantees that each intermediate path solution contributes to final query answers.

Algorithm 6 getNext of TJFastTL and GTJFastTL.

/* This following algorithm is used to evaluate query with only *parent–child* edge. */

Function getNext(n)

```
1: if (isLeaf(n)) then
2:      return n
3: else
4:   for each n_i ∈ dbl(n) do
5:      f_i = getNext(n_i)
6:      if (isBranching(n_i) ∧ empty(S_{n_i}))
7:         return f_i
8:      e_i = MB(n_i, n)
9: end for
```

```
10:   max = $maxarg_i\{ei\}$
11:   min = $minarg_i\{ei\}$
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:   **if** $(e_{min} == e_{max})$ updateSet($S_n$, $e_{min}$)
18:   return $f_{min}$
19: **end if**
```

6.3.2. Query with parent–child and ancestor–descendant edges

To evaluate a query with both *parent–child* and *ancestor–descendant* edges based on *tag + level* partition, we *simulate* multiple streams with the same tag to one *sorted* stream to reuse TJFast (and GTJFast) algorithm. In particular, our idea is that during query pattern matching, 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 maintaining a *min-heap* data structure to efficiently retrieve their minimal one. Therefore, TJFast (and GTJFast) can be reused to evaluate query in this new scenario.

6.4. Analysis of TJFastTL and GTJFastTL

In the section, we show the correctness of *TJFastTL* and *GTJFastTL* to analyze the efficiency.

Theorem 6.1. *Given an XML tree query Q and an XML database D, Algorithm TJFastTL and GTJFastTL correctly returns all answers for Q on D.*

Proof. There are two cases. (1) When queries contain only parent–child edges, we use Algorithm 6 to evaluate it. Here the key difference between TJFastTL (GTJFastTL) and TJFast (GTJFast) is in line 17 of getNext, where TJFastTL (GTJFastTL) 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 partition, 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_0 such that e_0 is an ancestor of e_{max} and e_0 involves in query answers. (2) The second case is that when query contains both parent–child and ancestor–descendant edges, we reuse TJFast (GTJFast) by simulating multiple streams with the same tag to one sorted stream. The correctness is obvious if the original algorithms are correct. □

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

Theorem 6.2. *Consider an XML database D and a tree query Q with (i) only parent–child relationships in all edges or (ii) only ancestor–descendant relationships between branching nodes and their children. The worst case I/O complexity of TJFastTL (and GTJFastTL) is linear in the sum of the sizes of input and output lists.*

It is worthy to note two advantages of algorithms based on tag + level over ones based on the traditional scheme: (1) *TJFastTL* and *GTJFastTL* typically scan less elements than TJFast and GTJFast by using *level pruning*; and (2) *TJFastTL* (*GTJFast*) shows a larger query class to guarantee the I/O optimality than TJFast (*GTJFastTL*).

7. Experimental evaluation

7.1. Experimental setup

7.1.1. Testbed and data set

We implemented eight XML tree pattern matching algorithms: TJFast, TJFastTL, GTJFast, GTJFastTL, TwigStack [5], TwigStackList [26], iTwigJoin [10] and *Twig*^{2}*Stack* [8] in JDK 1.4 using the file system as a simple storage engine. TJFast, TJFastTL, GTJFast and GTJFastTL, which are novel algorithms proposed in this article, are based on *extended Dewey* labeling scheme, and the other four use *region encoding* labeling scheme.

The reason that we choose these four existing algorithms for comparisons is that TwigStack, TwigStackList, iTwigJoin and *Twig*^{2}*Stack* are efficient for different applications. TwigStack [5] is very efficient when query contains only *ancestor–descendant* relationships. TwigStackList [26] is efficient on answering queries with *parent–child* relationships. Unlike the above two algorithms, which partition elements to one stream according to their tags alone, iTwigJoin [10] is a general twig join algorithm, which can be used on different data partition approaches. Ref. [10] used two data partitions: *tag + level* and *prefix path streaming*

Table 2
XML data sets.

	XMark	Random	DBLP	TreeBank
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

(PPS). Such *refined* data partition strategies enable iTwigJoin to reduce I/O cost by pruning irrelevant data streams. Finally, *Twig*^{2}*Stack* [8] is proposed to process generalized XML tree pattern queries.

All experiments were run on a 1.7G Pentium IV processor with 768 MB of main memory and 2 GB quota of disk space, running windows XP system. We use four different datasets including two synthetic and two real datasets [40]. These datasets differ from each other in terms of structure-complexity, data-size and data-distribution. Our goal in choosing these diverse sources is to understand the effectiveness and efficiency of our algorithms in different real world environments. Table 2 summarizes the characteristics of each data set.

XMark The first synthetic data is the well-known XMark benchmark data [42] (with factor 5), which describes information for an Internet auction website. The data set has about 8 million nodes and the average depth is 5.

TreeBank We obtained the TreeBank data set from the University of Washington XML repository [16], which annotates naturally-occurring text for linguistic structure. Tags in Treebank are recursive and highly nested. The data set has the maximal depth 36 and more than 2.4 million nodes.

Random We generated random data trees using two parameters: fan-out, depth. The fan-out of nodes in data trees uniformly vary in the range of 2–100. The depths of data trees vary from 10 to 100. We use ten different labels, namely: A_1,A_2,...,A_{10} to generate the data sets. The node labels in the trees are uniformly distributed. The random data has the average depth 5.1 and contains more than 5.1 million nodes.

DBLP DBLP data is a well-known public data set, which contains bibliographical data and has relatively flat structure. The average depth is only 2.9.

7.1.2. 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 encoding as an efficient way to present the integer value, which was proposed by Tatarinov et al. [37]. 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.

7.1.3. Labels size

We compare the labels size of four labeling schemes in Table 3. From this table, 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 helps to improve the performance of XML twig pattern matching.

7.2. Performance analysis

7.2.1. 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[2] shown in Table 4. Fig. 11(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 Fig. 11(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).

[2] We choose these queries according to XMark benchmark queries in [42].

Table 3
Labels size.

	XMark	Random	DBLP	TreeBank
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

Table 4
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

In order to research the effect of query path length on TJFast and PathStack, we then use 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 11(d–f) shows the execution times of both techniques, as well as the number of elements read and the size of disk files. Clearly, TJFast achieves 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.

7.2.2. Twig query pattern

We now focus on twig queries, and compare six holistic twig join algorithms TwigStack, TwigStackList, iTwigJoin, *Twig*^{2}*Stack* and TJFast, TJFastTL. We tested several XML queries on DBLP and TreeBank data (see Table 5). We chose these two data sets, because they are two extremes of the spectrum in terms of the structural complexity, that is, DBLP is a highly regular and shallow data, but Treebank is a highly irregular and deep data. The tested 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 contains both *parent–child* and *ancestor–descendant* relationships to connect the branching node.

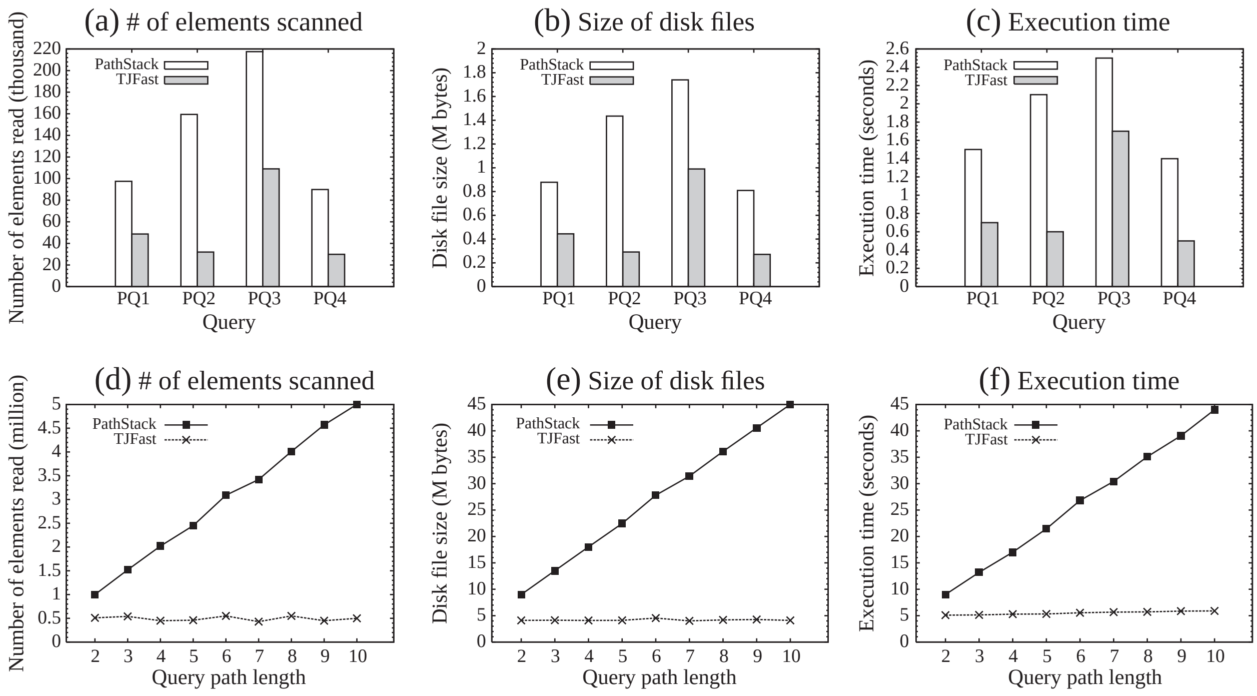


Fig. 11. PathStack versus TJFast using (a–c) XMark data and (d–f) random data.

Table 5
Twig queries on DBLP and TreeBank.

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

7.2.2.1. TJFast vs. TwigStack. We first compare the performance between TJFast and TwigStack. From Fig. 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.*

7.2.2.1.1. Cost of disk access. Fig. 12(a) shows that TJFast read far fewer elements than TwigStack. For example, in TQ1, TwigStack read 442,167 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.

7.2.2.1.2. Size of intermediate results. Table 6 shows the number of intermediate path solutions. 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 702,391 intermediate paths, while

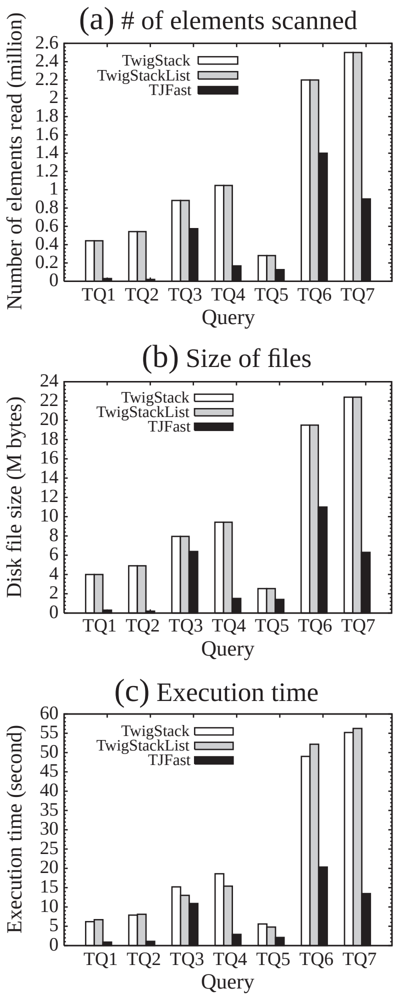


Fig. 12. TwigStack,TwigStackList vs. TJFast.

Table 6
Number of intermediate path solutions.

Query	TwigStack	TwigStackList	TJFast	TJFastTL	Useful
TQ_3	702,391	22,565	22,565	22,565	22,565
TQ_4	2237	388	316	302	302
TQ_5	10,663	9	9	9	5

only 22,565 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 22,565, which equals the number of useful solutions.

7.2.2.2. TJFast vs. TwigStackList. For all queries, TJFast outperforms TwigStackList again (see Fig. 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.

7.2.2.3. TJFast vs. iTwigJoin. We now compare the performance between TJFast and iTwigJoin. iTwigJoin is also based on *region encoding* labeling scheme, but it can be applied on different data partition strategies. Since [10] used two data partition strategies: *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 Fig. 13. As shown in this figure, 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.

Interestingly, the performance of iTwigJoin-PPS is fairly bad for all queries in TreeBank data (i.e. TQ3, TQ4 and TQ5). This can be explained that TreeBank is a deep and irregular data, which leads to a great number of streams and thus significantly increases the CPU cost. This result shows that PPS partition is not suitable for deep data set. This also explains the reason why we do not combine PPS data partition with our *extended Dewey* labeling scheme in this paper.

7.2.2.4. TJFast vs. TJFastTL. We now compare the performance between TJFast and TJFastTL. TJFastTL is based on *extended Dewey* labeling scheme and it uses *tag* + *level* data partition strategy. Fig. 14(a) shows that TJFastTL always read fewer elements than TJFast for seven twig queries. This is because TJFastTL uses level information to prune some useless streams. For example, in TQ4,

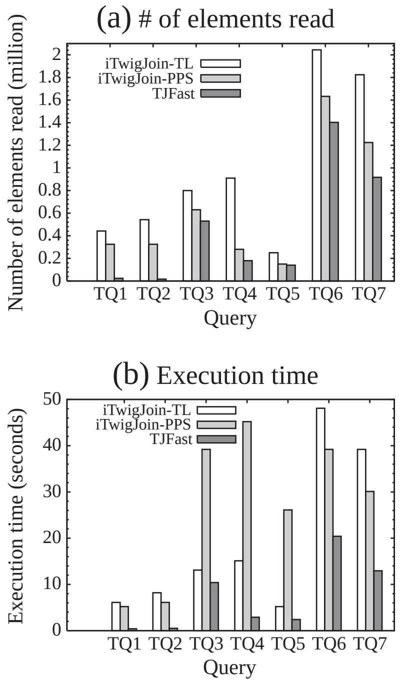


Fig. 13. iTwigJoin vs. TJFast.

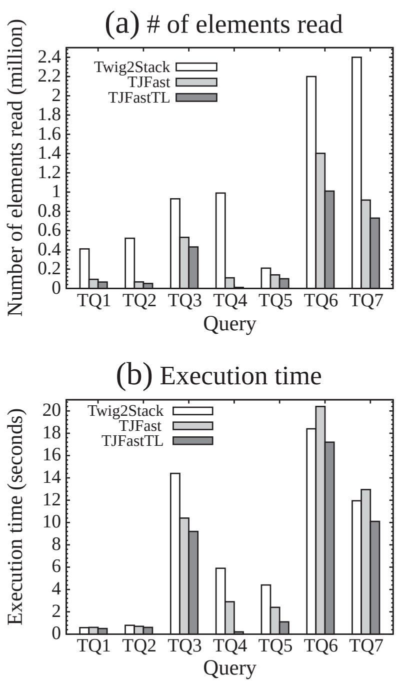


Fig. 14. Twig2Stack, TJFast and TJFastTL.

TJFastTL accesses only 736 elements, but TJFast needs to access 115,864 elements. This also explains the better performance of TJFastTL than that of TJFast in Fig. 14(b).

As mentioned in Theorem 6.2, unlike TJFast, TJFastTL is optimal for queries with only *parent–child* edges. This result can be confirmed by TQ4 in Table 6, which contains only *parent–child* relationships in all edges. For this query, TJFast outputs 316 intermediate path solutions, but TJFastTL outputs only 302. Notice that, for TQ4, the number of final useful path solutions is also 302. This observation shows that unlike TJFast, TJFastTL is an I/O optimal algorithm for TQ4.

7.2.2.5. TJFast, TJFastTL and Twig^{2}Stack. As the final experiment in this section, we compare the performance among TJFast, TJFastTL and *Twig^{2}Stack*. *Twig^{2}Stack* generalizes TwigStack algorithm to handle queries by using a novel hierarchical stack encoding to represent the twig results in main memory. Fig. 14(a) shows that TJFast and TJFastTL always read fewer elements than *Twig^{2}Stack* for seven twig queries, which is easily to understand, as *Twig^{2}Stack* use the same labeling scheme as TwigStack. Fig. 14(b) shows the evaluation results of three algorithms with respect to the execution time. Interestingly, we find that the performances between TJFast and *Twig^{2}Stack* are close (which is consistent with the results in [8]), but TJFastTL is always the winner in this competition. *Twig^{2}Stack* reduces the path merging cost using the compact hierarchal encoding, but suffers from the large input cost, TJFast saves input cost since it only needs to access the elements corresponding to the leaf query nodes. But TJFastTL is always the best one as it prunes away many irrelevant elements to significantly reduce I/O cost.

Note that TJFast and TJFastTL algorithms developed in this article are dedicated for reducing I/O cost of query processing. It is possible to extend both algorithms to further explore the available main memory to buffer more intermediate results for optimization. An optimized set encoding to compactly contain intermediate results is certainly a promising future direction to explore.

7.2.3. Generalized XML Tree queries

We now make experiments on generalized XML tree pattern queries, and compare three holistic join algorithms GTJFast, GTJFastTL and *Twig^{2}Stack* [8]. We tested six XML queries on XMark and TreeBank data (see Fig. 15). The tested queries have different generalized XML tree structures and combinations of return nodes, optional return nodes, mandatory axis and optional axis.

7.2.3.1. GTJFast vs. GTJFastTL. We now compare the performance between GTJFast and GTJFastTL. GTJFastTL is based on *extended Dewey* labeling scheme and it uses *tag* + *level* data partition strategy. Fig. 16(a) shows that GTJFastTL always reads fewer elements than GTJFast for six twig queries. This is because TJFastTL uses level information to prune some useless streams.

7.2.3.2. GTJFastTL vs. Twig^{2}Stack. Finally, we compare GTJFastTL with *Twig^{2}Stack*, which is a recent proposed algorithm for generalized tree pattern. As shown in Fig. 16, GTJFastTL achieves better performance than *Twig^{2}Stack*. GTJFastTL maintains the

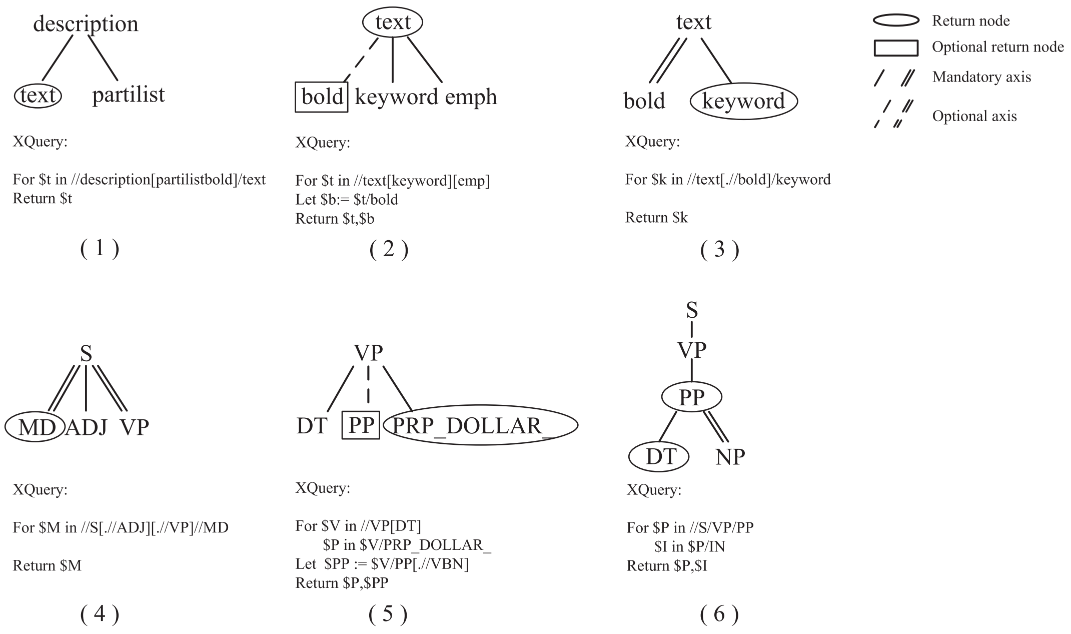


Fig. 15. Six generalized XML tree patterns.

BitArray in the main memory and avoids the output of useless results. In addition, the level pruning in GTJFastTL avoids the scan of many useless elements.

7.2.4. Summary

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

1. 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

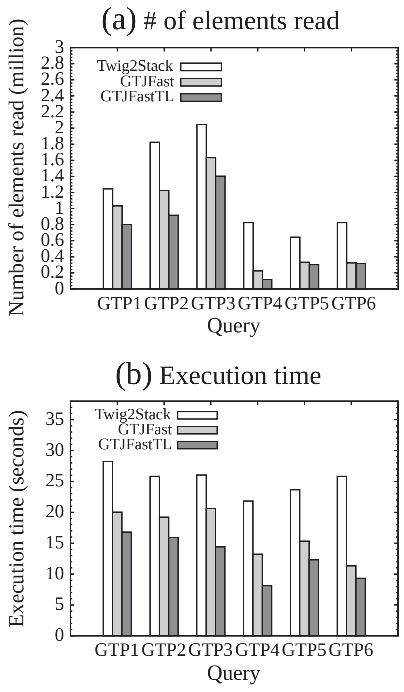


Fig. 16. GTJFast, GTJFastTL and *Twig^{2}Stack*.

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.
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. Generalized tree pattern matching algorithms, GTJFast and GTJFastTL, exploit the property of non-return nodes and use the BitArray to maintain the intermediate matching results to significantly reduce the output of useless path solutions.

8. Related work

Labeling schemes *Dewey ID* labeling scheme comes from the work of Tatarinov et al. [37] to represent XML order in the relational data model, and to show how this labeling scheme can be used to preserve document order during XML query processing. O'Neil et al. [32] 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 main idea of ORDPATH is to use only positive, odd integers to label elements in an initial load and even and negative integers component values are reserved for later insertions into an existing tree.

The *region encoding* is considered as the work of Consens and Milo [13], who discussed a fragment of PAT text searching operators for indexing text database. Then Zhang et al. [47] introduced it to XML query processing using inverted list. Recently, many researchers [6,23,36,44,46,49] have begun to design a dynamic XML labeling scheme on the context of frequent inserting and deleting data.

Query processing algorithms Holistic XML matching algorithms are prevalent for matching pattern queries over stored XML data. They demonstrate good performance due to their ability to minimize unnecessary intermediate results. In particular, N. Bruno et al. [5] proposed a holistic twig join algorithm, namely TwigStack. Lu et al. [27] researched how to answer an *ordered* twig pattern based on *region encoding*. Chen et al. [10] 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. [17] proposed a general holistic 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. Jiang et al. [16] also studied the problem of processing queries with OR predicates. BLAS by Chen et al. [11] proposed a bi-labeling 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. [45] proposed the idea of the combination of path index table and Dewey labels. 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. Additionally, the recent works of Zhang et al. [48] and Koloniari et al. [18] begin to research the distributed XQuery and XPath processing.

The preliminary idea of extended Dewey labeling scheme and the TJFast algorithm appeared in [29]. We make the new contributions in this article by proposing three novel algorithms TJFastTL, GTJFast and GTJFastTL to handle generalized tree pattern queries with optimization, and providing comprehensive analysis and experiments to compare those proposed algorithms.

To analyze the space complexity of processing XML twig queries, Shalem et al. [35] showed that the upper bound of full-fledge queries (i.e. queries which only require to return output nodes) with P–C and A–D edges is $O(D)$, where D is the document size. Those results are consistent with our research, but our research in TJFast and GTJFast algorithms show a large sub-query class with A–D and P–C relationships to guarantee the small space cost which is quadratic to document depth.

Finally, ViST and PRIX [39,34] transform both XML data and queries into sequences and answer XML queries through subsequence matching. While their methods avoid join operations in query processing, to eliminate *false alarm* and *false dismissal*, they resort to post-processing(for *false alarm*) and multiple isomorphism queries processing[38](for *false dismissal*), both of which are time consuming.

9. Conclusions and future work

XML tree pattern matching is an important issue in XML query processing. In this article, we have proposed TJFast as an efficient algorithm to address this problem using a novel labeling scheme: *extended Dewey*. Although the idea of *Dewey* labeling scheme is not new, extending it to efficiently process XML twig pattern matching is nontrivial. This is because based on the *Dewey* labeling scheme, we cannot know the element names along a path. To answer a tree 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, TJFast only needs to access labels of *leaf* nodes to answer queries and significantly reduce I/O cost. Furthermore, to efficiently evaluate generalized XML tree patterns, we propose GTJFast to compactly represent the intermediate matching results and avoid the output of non-return nodes. In addition, this article also makes the contribution by proposing the algorithm TJFastTL and GTJFastTL based on *tag* + *level* data partition strategy to avoid the scan of useless elements. Finally, we made the comprehensive experimental results to show that our set of XML tree pattern matching algorithms are superior to existing approaches in terms of *the number of elements scanned*, *the size of intermediate results* and *query performance*.

Exciting follow-up research can be centered around improving efficiency of querying algorithms and labeling scheme. One interesting and intriguing problem is to handle the large size of labels after frequent skewed insertions for dynamic XML documents. A promising approach for solving the problem would be to encode each label using compact dynamic binary string (CDBS) in [25], which use binary strings to compactly represent the order and handle updates. Another topic of interest is to investigate how to use some index structures (e.g. a variant of B + tree) to further speedup the query processing based on *extended Dewey* labeling scheme. The main advantage of our algorithms proposed in this article is to access elements for only leaf query nodes and thus reduce I/O cost. We can leverage index structures to further reduce I/O cost by accessing *only part* of elements for leaf query nodes. But we should respect the trade-off that indexing data introduces more cost on index storage and internal nodes traversal. Our flexibility in this balance comes from our ability to create a smaller B + tree. The current preliminary idea is to mine twig query log to find the paths which are queried intensively and index those elements from frequent paths by B + tree. We treat the detailed pattern mining and index construction algorithms as our future work.

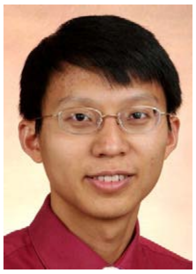
Acknowledgement

This paper was partially supported by 863 National High-Tech Research Plan of China (no. 2009AA01Z133), National Science Foundation of China (NSFC) (no. 60903056), SRFDP Fund for the Doctoral Program (no. 20090004120002), IBM Research-China Fund (Project no. JSA200909010), SRF for ROCS, SEM. and the Program for New Century Excellent Talents in University.

References

[1] Shurug Al-Khalifa, H.V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, Divesh Srivastava, Structural joins: a primitive for efficient XML query pattern matching, Proceedings of the 18th International Conference on Data Engineering (ICDE), San Jose, CA, USA, 2002, pp. 141–152.
[2] B. Baeza-Yates, G.H. Gonnet, A new approach to text searching, Proceedings of the 12th International Conference on Research and Development in Information Retrieval (SIGIR), Cambridge, Massachusetts, USA, 1992, pp. 168–175.
[3] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernandez, M. Kay, J. Robie, J. Simeon, XML path language (XPath) 2.0 W3C working draft 16, Tech. Rep. WDxpath20-20020816, World Wide Web Consortium, August 2002. <http://www.w3.org/TR/xpath20>.
[4] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, Jerome Simeon, XQuery 1.0: An XML Query Language W3C Recommendation 23 January 2007.
[5] Nicolas Bruno, Nick Koudas, Divesh Srivastava: Holistic twig joins: optimal XML pattern matching, in, Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD), Madison, Wisconsin, USA, 2002, pp. 310–321.
[6] Barbara Catania, Wen Qiang Wang, Beng Chin Ooi, Xiaoling Wang, Lazy XML updates: laziness as a virtue of update and structural join efficiency, Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, 2005, pp. 515–526.
[7] Y. Chi, X. Song, D. Zhou, K. Hino, B.L. Tseng, On evolutionary spectral clustering, ACM Trans. Knowl. Discov. Data, 3, 4, Nov. 2009, pp. 1–30, DOI= http://doi.acm.org/10.1145/1631162.1631165.
[8] Songting Chen, Hua-Gang Li, Jun'ichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, K. Seluk Candan, Twig2Stack: bottom-up processing of generalized-tree-pattern queries over XML documents, Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, 2006, pp. 283–294.
[9] Zhimin Chen, H.V. Jagadish, Laks V.S. Lakshmanan, Stelios Paparizos, From tree patterns to generalized tree patterns: on efficient evaluation of XQuery, Proceedings of 29th International Conference on Very Large Data Bases, Berlin, Germany, 2003, pp. 237–248.
[10] Ting Chen, Jiaheng Lu, Tok Wang Ling, On boosting holism in XML twig pattern matching using structural indexing techniques, Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), Baltimore, Maryland, USA, 2005, pp. 455–466.
[11] Yi. Chen, Susan B. Davidson, Yifeng Zheng, BLAS: an efficient XPath processing system, Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), Paris, France, 2004, pp. 47–58.
[12] Zhiyuan Chen, Johannes Gehrke, Flip Korn, Nick Koudas, Jayavel Shanmugasundaram, Divesh Srivastava, Index structures for matching XML twigs using relational query processors, Data Knowl. Eng. (DKE) 60 (2) (2007) 283–302.
[13] Mariano P. Consens, Tova Milo, Algebras for querying text regions, Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Jose, California, USA, 1995, pp. 11–22.
[14] R. Goldman, J. Widom, Dataguides: enabling query formulation and optimization in semistructured databases, Proceedings of the 23rd International Conference on Very Large Data Bases, August 25–29, 1997, 1997, pp. 436–445, Morgan Kaufmann, Athens, Greece.
[15] Theo Harder, Michael Peter Haustein, Christian Mathis, Markus Wagner, Node labeling schemes for dynamic XML documents reconsidered, Data Knowl. Eng. (DKE) 60 (1) (2007) 126–149.
[16] Haifeng Jiang, Hongjun Lu, Wei Wang, Efficient processing of twig queries with OR-predicates, Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, 2004, pp. 59–70.
[17] Haifeng Jiang, Wei Wang, Hongjun Lu, Jeffrey Xu. Yu, Holistic twig joins on indexed XML documents, Proceedings of 29th International Conference on Very Large Data Bases, 2003, pp. 273–284, Morgan Kaufmann, Berlin, Germany.
[18] G. Koloniari, E. Pitoura, Distributed structural relaxation of XPath queries, Proceedings of the 25th International Conference on Data Engineering, ICDE, Shanghai, China, 2009, pp. 529–540.
[19] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, Henry F. Korth, Covering indexes for branching path queries, Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, 2002, pp. 133–144.
[20] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, Raghu Ramakrishnan, On the integration of structure indexes and inverted lists, Proceedings of the 20th International Conference on Data Engineering, ICDE, Boston, MA, USA, 2004, pp. 829–831.
[21] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, Ehud Gudes, Exploiting local similarity for indexing paths in graph-structured data, Proceedings of the 18th International Conference on Data Engineering (ICDE), San Jose, CA, 2002, pp. 129–140.
[22] Donald E. Knuth, James H. Morris Jr., Vaughan R. Pratt, Fast pattern matching in strings, SIAM J. Comput. (SIAMCOMP) 6 (2) (1977) 323–350.
[23] A.J. Lee, H. Wu, T. Lee, Y. Liu, K. Chen, Mining closed patterns in multi-sequence time-series databases, Data Knowl. Eng. 68 (10) (Oct. 2009) 1071–1090.
[24] Quanzhong Li, Bongki Moon, Indexing and querying XML data for regular path expressions, Proceedings of 27th International Conference on Very Large Data Bases (VLDB), Roma, Italy, 2001, pp. 361–370.
[25] Changqing Li, Tok Wang Ling, Min Hu, Efficient updates in dynamic XML data: from binary string to quaternary string, VLDB J., 17(3), 2008, pp. 573–601.
[26] Jiaheng Lu, Ting Chen, Tok Wang Ling, Efficient processing of XML twig patterns with parent child edges: a look-ahead approach, Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management, Washington, DC, USA, 2004, pp. 533–542.
[27] Jiaheng Lu, Tok Wang Ling, Tian Yu, Changqing Li, Wei Ni, Efficient processing of ordered XML twig pattern, Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management (CIKM), Washington, DC, USA, 2004, pp. 300–309.
[28] Jiaheng Lu, Ting Chen, Tok Wang Ling, TJFast: effective processing of XML twig pattern matching, Proceedings of the WWW (Special interest tracks and posters), 2005, pp. 1118–1119.
[29] Jiaheng Lu, Tok Wang Ling, Chee Yong Chan, Ting Chen, From region encoding to extended Dewey: on efficient processing of XML twig pattern matching, VLDB, 2005, pp. 193–204.

[30] Tova Milo, Dan Suciu: Index Structures for Path Expressions, Proceedings of the 7th Database Theory International Conference (ICDT), Jerusalem, Israel, 1999, pp. 277–295.
[31] Chung Keung Poon, Leo Yuen, Faster twig pattern matching using extended Dewey ID, Proceedings of the Database and Expert Systems Applications, DEXA, Krakow, Poland, 2006, pp. 297–306.
[32] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, Nigel Westbury, ORDPATHs: insert-friendly XML node labels, Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, 2004, pp. 903–908.
[33] Lu. Qin, Jeffrey Xu. Yu, Bolin Ding, TwigList: make twig pattern matching fast, Proceedings of the 12th International Conference on Database Systems for Advanced Applications, DASFAA, Bangkok, Thailand, 2007, pp. 850–862.
[34] Praveen Rao, Bongki Moon, Indexing and querying XML using Pr¨¹fer sequences, Proceedings of the 20th International Conference on Data Engineering, ICDE, Boston, MA, USA, 2004, pp. 288–300.
[35] Mirit Shalem, Ziv Bar-Yossef, The space complexity of processing XML twig queries over indexed documents, Proceedings of the 24th International Conference on Data Engineering, ICDE, Cancun, Mexico, 2008, pp. 824–832.
[36] S. Soltan, A. Zarnani, R. AliMohammadzadeh, M. Rahgozar, "IFDewey: A New Insert-Friendly Labeling Schema for XML Data", World Academy of Science, Engineering and Technology, 2006, pp. 116–119.
[37] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, Chun Zhang, Storing and querying ordered XML using a relational database system, Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, 2002, pp. 204–215.
[38] Haixun Wang, Xiaofeng Meng, On the sequencing of tree structures for XML indexing, Proceedings of the 21st International Conference on Data Engineering, ICDE, Tokyo, Japan, 2005, pp. 372–383.
[39] Haixun Wang, Sanghyun Park, Wei Fan, Philip S. Yu, ViST: A Dynamic Index Method for Querying XML Data by Tree Structures, in Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, SIGMOD pp 110–121.
[40] University of Washington XML repository http://www.cs.washington.edu/research/xmldatasets/.
[41] Fangju Wang, Jing Li, Hooman Homayounfar, A space efficient XML DOM parser, Data Knowl. Eng. (DKE) 60 (1) (2007) 185–207.
[42] Xmark ¡ª an XML benchmark project. http://monetdb.cwi.nl/xml/index.html.
[43] Xin Wu, Guiquan Liu, XML twig pattern matching using version tree, Data Knowl. Eng. (DKE) 64 (3) (2008) 580–599.
[44] Xiaodong Wu, Mong-Li Lee, Wynne Hsu, A prime number labeling scheme for dynamic ordered XML trees, Proceedings of the 20th International Conference on Data Engineering, ICDE, Boston, MA, USA, 2004, pp. 66–78.
[45] Beverly Yang, Marcus Fontoura, Eugene J. Shekita, Sridhar Rajagopalan, Kevin S. Beyer, Virtual cursors for XML joins, in: Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management, Washington, DC, USA pp 523–532.
[46] Jung-Hee Yun, Chin-Wan Chung, Dynamic interval-based labeling scheme for efficient XML query and update processing, J. Syst. Softw. (JSS) 81 (1) (2008) 56–70.
[47] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, Guy M. Lohman: On Supporting Containment Queries in Relational Database Management Systems, in: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, pp 425–436.
[48] Ying Zhang, Nan Tang, Peter Boncz, Efficient distribution of full-fledged XQuery, Proceedings of the 25th International Conference on Data Engineering, ICDE, Shanghai, China, 2009, pp. 565–576.
[49] L. Zhu, W. Meng, W. Yang, C. Liu, Region clustering based evaluation of multiple top-N selection queries, Data Knowl. Eng., 64, 2, Feb. 2008, pp. 439–461.



Jiaheng Lu is an associate professor at the Renmin University of China. He got his Ph.D. degree at the National University of Singapore. His research interests include XML data management, keyword search and cloud data management. He has published more than 20 papers in top conferences and journals. He has served as a program member in database conferences including SIGMOD, VLDB and ICDE.

Xiaofeng Meng is a full Professor and Vice Dean at the School of Information, Renmin University of China. He received his Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, all in computer science. His research interests include web data integration, native XML databases, cloud data management, moving object management, flash based database systems, data privacy and trust. He has published over 100 papers in refereed international journals and conference proceedings.

Tok Wang Ling received his Ph.D. in Computer Science from the University of Waterloo (Canada). He is a professor of the Department of Computer Science at the National University of Singapore. His research interests include Data Modeling, ER approach, Normalization Theory, and Semistructured Data Model and XML query processing. He has published more than 180 papers, co-authored a book, and co-edited 9 conference proceedings. He is an ACM Distinguished Scientist and a senior member of IEEE.