Journal of Zhejiang University SCIENCE A
ISSN 1673-565X (Print); ISSN 1862-1775 (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Efficient processing of ordered XML twig pattern matching based on extended Dewey*

Jin-hua JIANG†, Ke CHEN, Xiao-yan LI, Gang CHEN, Li-dan SHOU
(*School of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China*)
†E-mail: jiangjinhua.zju@163.com

Received Jan. 2, 2009; Revision accepted Apr. 15, 2009; Crosschecked Oct. 18, 2009

Abstract: Finding all occurrences of a twig pattern is a core operation of extensible markup language (XML) query processing. Holistic twig join algorithms, which avoid a large number of intermediate results, represent the state-of-the-art algorithms. However, ordered XML twig join is mentioned rarely in the literature and previous algorithms developed in attempts to solve the problem of ordered twig pattern (OTP) matching have poor performance. In this paper, we first propose a novel children linked stacks encoding scheme to represent compactly the partial ordered twig join results. Based on this encoding scheme and extended Dewey, we design a novel holistic OTP matching algorithm, called OTJFast, which needs only to access the labels of the leaf query nodes. Furthermore, we propose a new algorithm, named OTJFaster, incorporating three effective optimization rules to avoid unnecessary computations. This works well on available indices (such as B^+-tree), skipping useless elements. Thus, not only is disk access reduced greatly, but also many unnecessary computations are avoided. Finally, our extensive experiments over both real and synthetic datasets indicate that our algorithms are superior to previous approaches.

Key words: XML querying, Ordered twig join, Index, Optimization
doi:10.1631/jzus.A0920006 **Document code:** A **CLC number:** TP311.13

INTRODUCTION

Because of its semantic relevance, scalability and standardization, XML is widely used in digital library, e-commerce, web and other applications, and it has become the de facto standard for information representation and data exchange. The problem of query processing of XML data has been among the major issues of database research. Most XML query languages such as XPath (Berglund *et al*., 2002) and XQuery (Boag *et al*., 2002) allow users to run queries against the XML document in the form of twig patterns, which typically specify patterns of selection predicates on multiple elements (Jiang HF *et al*., 2003).

Finding all occurrences of a twig pattern is a core operation of XML query processing (Al-Khalifa *et al*., 2002; Bruno *et al*., 2002; Jiang *et al*., 2003; Chen *et al*., 2005; Lu *et al*., 2005a). Many algorithms have recently been proposed to process the twig pattern matching efficiently. The holistic twig join algorithms (Bruno *et al*., 2002; Jiang HF *et al*., 2003; Lu *et al*., 2004; 2005a; Chen *et al*., 2006; Jiang ZW *et al*., 2006) have demonstrated superior performance, avoiding creating a large number of intermediate results.

However, both XPath and XQuery also define four order navigation axes: following-sibling, following, preceding-sibling and preceding (Lu *et al*., 2005b; Zografoula *et al*., 2005). We denote a query pattern with order axes as an ordered twig pattern (OTP). For example, the query '//inproceedings/author/following-sibling::title' means to find all 'titles' that are following siblings of the 'author' that should be a child of 'inproceedings'. Supporting the efficient processing of OTP matching is a key issue in

* Project supported by the National Natural Science Foundation of China (Nos. 60603044 and 60803003), the Program for the Changjiang Scholars and Innovative Research Team in University (No. IRT0652), and the Key Technology Projects of Zhejiang Province, China (No. 2006c11108)

XML data management, but has been rarely mentioned in previous work. To handle an OTP matching, a straightforward approach that first matches the unordered twig pattern and then prunes away the undesired answers is obviously not optimal because of the large number of intermediate results. Three algorithms for XML OTP matching have been published recently: PRIX (Rao and Moon, 2004; 2006), OrderedTJ (Lu *et al*., 2005b) and LBHJ (Zhu *et al*., 2008). Next, we briefly describe these three algorithms and discuss some of their drawbacks that motivated us to develop our proposed approaches.

PRIX Based on Prűfer (1918)'s method that constructs a one-to-one correspondence between trees and sequences by using the node removal method, PRIX (Rao and Moon, 2004; 2006) allows holistic processing of OTP matching by performing subsequence matching. In particular, it supports ordered twig queries inherently. However, it resorts to post-processing and special operators for refinement, which is time-consuming. In addition, when the document has a recursive deep structure, the subsequence matching is inefficient. Furthermore, as shown in Example 1, when the query is not the subgraph of the XML document, PRIX will work with false dismissals.

Example 1 Consider the XML document tree *T* in Fig.1a and two twig queries Q1 and Q2 in Figs.1b and 1c. Tree *T* has LPS(*T*)={C B D B A E A} such that LPS denotes the labeled Prűfer sequence (Rao and Moon, 2004). As Q1 is a subgraph of tree *T*, LPS(Q1)={C B D B A} matches a subsequence of LPS(*T*) at positions 1, 2, 3, 4, 5. However, although clearly there is a match of Q2 in tree *T*, LPS(Q2)={C A D A} does not have a match of subsequence of LPS(*T*).

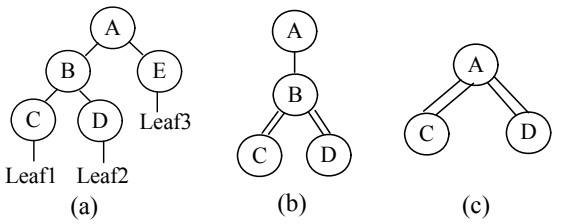

Fig.1 A sample XML document and two twig queries
(a) Tree *T*; (b) Q1; (c) Q2

OrderedTJ and LBHJ Both OrderedTJ (Lu *et al*., 2005b) and LBHJ (Zhu *et al*., 2008) were proposed to process XML OTP matching in a holistic way. However, they need to access the labels of all query nodes, which is time-consuming and inefficient. Moreover, LBHJ can process OTP with following-sibling or preceding-sibling relationships directly, while the following and preceding axes are out of its consideration.

In this paper, we address the problem of efficient holistic processing of OTP matching. We propose a novel algorithm, named OTJFast (ordered twig join fast), which needs only to access the labels of the leaf query nodes. Thus, both the disk access and the CPU cost can be greatly reduced. A preliminary version of our work has been published (Jiang JH *et al.*, 2008). In this paper, we extend the research by adding detailed analyses, introducing a novel algorithm (OTJFaster) and conducting new comprehensive experiments.

Our contributions in this paper are summarized as follows:

1. We propose the children linked stacks encoding scheme, which compactly represents the partial OTP matching result, with the full analyses and proofs.

2. Based on this encoding scheme, we propose an efficient holistic OTP matching algorithm, called OTJFast, which supports OTP with all of the four order axes and wildcard '*'.

3. We introduce a new algorithm, named OTJFaster, incorporating three optimization rules to explore opportunities to skip useless elements. Our algorithm works well with available indices such as B^+-tree. Thus, not only do we reduce disk access, but also we skip a large number of useless recursive calls.

4. We implement our algorithms and perform comprehensive experiments over both real and synthetic datasets. The results show the advantages of our algorithms over previous approaches for XML OTP matching.

BACKGROUND

Data model and labeling scheme

An XML document is commonly modeled as a labeled, ordered, nested structure tree, where each node corresponds to an element, an attribute or a text value. The parent-child pairs represent nesting relationships between XML elements, and the ordering of sibling nodes implicitly defines a total order on the

nodes. We use the extended Dewey (Lu *et al*., 2005a) to encode the XML elements. Using this approach, not only can the structure relationships be derived by string matching but also the element names along the path from the root to the element can be derived using a finite state transducer (FST) (Lu *et al*., 2005a).

Example 2 Fig.2a shows an XML tree such that each element in the tree is associated with an extended Dewey. The DTD and the FST for the DTD are shown in Figs.2b and 2c, respectively. For instance, given an extended Dewey label '0.1.0.6', its path derived from the label should be 'a/c/b/d' by using the FST in Fig.2c, transmitting from the initial state *a* to state *d* by modulo function.

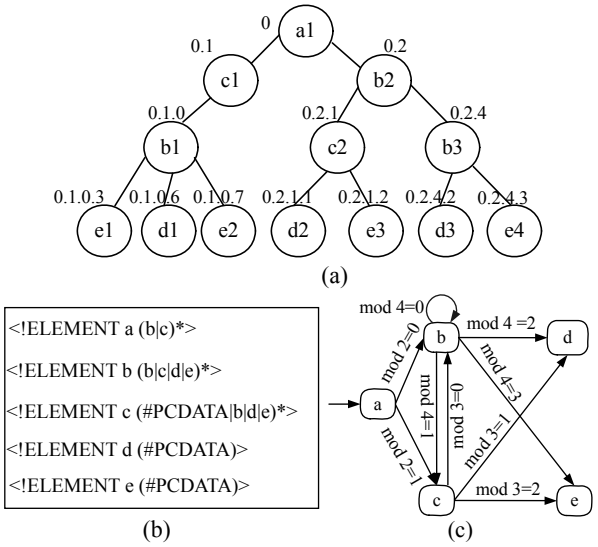

Fig.2 (a) An XML tree; (b) The DTD; (c) Finite state transducer (FST) for the DTD

Ordered twig pattern matching

An OTP can also be represented as an ordered, labeled tree. Each node of an OTP may be an element tag, a text value or a wildcard '*' (Lu *et al*., 2005a). The pattern edges can be parent-child (P-C) relationships, ancestor-descendant (A-D) relationships, following relationships or following-sibling relationships. Given an OTP Q, we use Q_n to denote the subtree rooted with n.

Fig.3 shows two sample XPaths (Fig.3a) and their corresponding OTPs (Figs.3b and 3c). We use the symbol '>' in a box to mark a branching node whose children nodes should appear in order.

Note that we can transform a preceding (preceding-sibling) relationship into an equivalent following (following-sibling) relationship. Moreover, once there are P-C relationships in branching edges, we translate them into A-D relationships and then test for the P-C relationships during the merging phase. First, we deal with the problem of finding all occurrences of XML OTP without following-sibling relationships or wildcards '*'. Later, in the subsection "Processing the following-sibling relationship and wildcard '*'", we will explain how OTP with following-sibling axes and wildcards '*' can be matched.

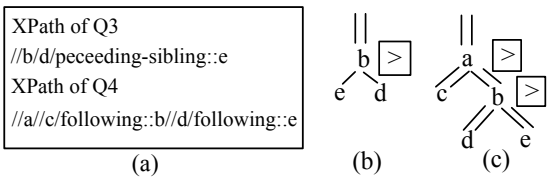

Fig.3 (a) Two XPaths; (b), (c) The corresponding ordered twig patterns

To distinguish between a twig pattern tree and an XML document tree, in the rest of this paper, 'node' refers to a tree node in the twig pattern, while 'element' refers to an element in an XML document. We use P_n to denote the path from the root to n for any node n in the query tree, and the path P_e for element e in the XML document is defined in the same way. Note that given an element e, we can derive its path P_e from its extended Dewey.

Formally, given an OTP Q and an XML document D, a match of Q in D can be identified by a mapping from nodes in Q to the elements in D, such that (1) the query node predicates, (2) the P-C and A-D relationships between query nodes, and (3) the order of the query sibling nodes, are satisfied by the corresponding elements. The answers to a query Q with n nodes can be represented as a list of n-ary tuples, where each tuple $<v_1,v_2,\ldots,v_n>$ consists of the elements that identify a distinct match of Q in D. For example, the result of Q3 in Fig.3b for the XML document in Fig.2a contains only <b1,d1,e1>, and does not contain <b1,d1,e2> or <b3,d3,e4>.

Notations

In our paper, following TJFast, we make use of the following functions over query nodes. isLeaf(n) and isBranching(n) return whether the query node n is a leaf or a branching node. leafNodes(n) returns the set of leaf nodes in Q_n. directBranchingOrLeafNodes(n) (for short, dbl(n)) returns the set of all branching nodes b and leaf nodes f in Q_n such that in

the path from n to b or f (excluding n, b or f) there are no branching nodes (Lu *et al*., 2005a). Furthermore, for OTP matching, we add two additional functions over query nodes: children(n) and rightSibling(n), which return the set of the children nodes of n and the immediate right sibling node that shares the common parent node with n, respectively.

Associated with each leaf node f in an OTP there is a stream T_f. Each T_f contains extended Dewey labels of elements that match the node type f. The elements in the stream are sorted by ascending lexicographical order. Three functions current(T_f), advance(T_f) and eof(T_f) over the stream T_f are defined, such that current(T_f) returns the current extended Dewey label of the stream, advance(T_f) updates the current element of the stream to be the next element, and eof(T_f) judges whether the stream T_f reaches the end.

There are three self-explanation operations defined over elements in the XML document: ancestors(e), descendants(e) and isRight(e_a, e_b). The functions ancestors(e) and descendants(e) return the ancestors and descendants, respectively, of e (both including e). Given two elements e_a and e_b, the function isRight(e_a, e_b) determines whether e_b follows e_a. In particular, within extended Dewey, it means $P_{ea}<P_{eb}$ by ascending lexicographical order and P_{ea} is not a prefix of P_{eb}.

CHILDREN LINKED STACKS ENCODING

In this section, we propose the children linked stacks encoding scheme, which is an important concept in our algorithms. The analyses and the proofs are described in detail.

Motivations

To process the ordered twig queries efficiently, there are two main principles. First, we try to access only the labels of the leaf query nodes by using the powerful labeling scheme extended Dewey. Second, we need an effective mechanism to represent partial ordered twig results compactly to minimize the intermediate results. Based on these principles, we propose the children linked stacks encoding scheme (CLS), where we cache the elements that match the children nodes of the branching node in a linked stack to capture and check the order relationships between the elements.

Data structures

Given a branching node n, as well as the normal attached stack S_n (Lu *et al*., 2005a), we associate it with children linked stacks CLS[n], which consist of a linked ordered sequence of stacks. Each stack CLS$_i$[n] of CLS[n] ($1\le i\le$|children(n)|) contains several data nodes. Each data node in CLS$_i$[n] consists of a pair: (1) an element e_i, which matches the ith child node n_i of n, with its extended Dewey label and (2) a pointer that points to a node in the CLS$_{i+1}$[n] (initially the pointer is null). At every point during the computation, the nodes in each stack CLS$_i$[n] (from bottom to top) lie on the same root-leaf path in ascending lexicographical order according to their extended Dewey labels. All CLS$_i$[n] of CLS[n] are linked together by the pointers of the data nodes, hence the term linked stacks.

Example 3 Given Q4 in Fig.3c for the data in Fig.2a, when the leaf elements c2, d3, e4 are scanned, Fig.4 shows the normal stacks and the CLSs of query nodes a and b. The creation of the CLSs will be explained in the subsection 'Creating and checking children linked stacks'.

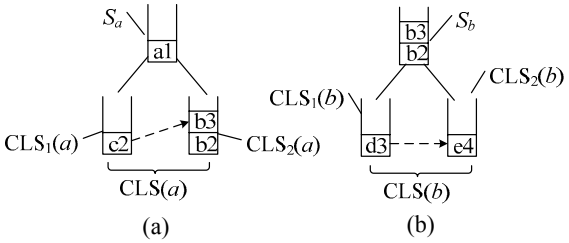

Fig.4 Normal stacks and CLSs of nodes *a* (a) and *b* (b)

Creating and checking children linked stacks

Algorithm 1 shows the process of creating and checking children linked stacks, where we accomplish two tasks. The first is to fill each stack of CLS[n] with the corresponding data nodes (lines 2~11). The second task is to set the pointers of the data nodes in each stack and link the CLS$_i$ together to check the order of the elements (line 12).

For the first task, we fill each stack CLS$_i$[n] of CLS[n], which corresponds to the ith child node n_i of n, with the data nodes according to the following criteria: (1) If n_i is a branching node, we copy the elements of the normal stack S_{ni} to stack CLS$_i$[n] (lines 3~5). Note that we do the twig joins in leaf-to-root order, and thus the elements in the stack S_{ni} match the branching node n_i for Q_{ni} (detailed explanations are given in the subsection of 'The OTJFast

algorithm'). (2) If n_i is a leaf node, the current element of T_{ni} is inserted into the stack $CLS_i[n]$ (lines 6 and 7). (3) Otherwise, we update the stack $CLS_i[n]$ with the elements that match the node n_i in the path solution to path pattern $dbl(n)_i$ under the element e (lines 8~11).

Algorithm 1 createCheckCLS(n,e)

```
1 for each n_i∈children(n) do
2     CLS_i[n]=the ith stack of CLS[n];
3     if isBranching(n_i) then
4         for each e_i∈S_ni do
5             updateStack(CLS_i[n],e_i);
6     else if isLeaf(n_i) then
7         updateStack(CLS_i[n],current(n_i));
8     else
9         for each e_i∈MB(dbl(n)_i,n_i) do
10            if e∈ancestors(e_i) then
11                updateStack(CLS_i[n],e_i);
12 return setAndCheckLink(n,e);

Procedure clearStack(S,e)
   Deleting any element a in the stack S such
     that a∉ancestors(e) and e∉ancestors(a);

Procedure updateStack(S,e)
1  clearStack(S,e);
2  Add e to stack S;
```

For the second task (Algorithm 2), the pointer of the data node d_i (with element e_i) in $CLS_i[n]$ is assigned to point to the data node d_{i+1} with the minimal element e_{i+1} in $CLS_{i+1}[n]$ such that e_{i+1} follows e_i, if any (lines 2 and 3). Once there is a stack $CLS_i[n]$ whose data nodes are not associated with any pointers, the index i is returned (lines 4 and 5). Otherwise, we return the result of checkLink(n,1,1) (line 6).

Example 4 Now we explain how we create the CLSs in Example 3. Given that the leaf elements c2, d3, e4 are scanned, following TJFast, then elements b2 and b3 are inserted into the normal stack S_b and a1 is inserted into S_a. Since node d and node e are leaf nodes in Q4 in Fig.3c, when createCheckCLS(b, b2) is called, d3 is inserted into $CLS_1[b]$ and e4 is inserted into $CLS_2[b]$ (lines 6 and 7 in Algorithm 1). Then setAndCheckLink(b, b2) will be called in line 12 (Algorithm 1). Note that e4 is the minimal element in $CLS_2[b]$ of $CLS[b]$ such that P_{e4} follows P_{d3}. The pointer of d3 in $CLS_1[b]$ is assigned to point towards e4 in $CLS_2[b]$ (line 3 in Algorithm 2). When createCheckCLS(a,a1) is called, as c is a leaf node, c2 is inserted into $CLS_1[a]$ of $CLS[a]$. Note that b is a branching node. We copy the elements b2 and b3 in the normal stack S_b into $CLS_2[a]$ (lines 3~5 in Algorithm 1). Finally, createCheckCLS(a,a1) calls setAndCheckLink(a,a1). As P_{b3} follows P_{c2} while P_{b2} does not, we make the pointer of c2 in $CLS_1[a]$ point to b3 in $CLS_2[a]$.

Algorithm 2 setAndCheckLink(n,e)

```
1 for each CLS_i∈CLS[n] do
2     for each d_i in CLS_i do
3         make the pointer of d_i with element e_i point
            to data node d_{i+1} with the minimal element e_{i+1}
            in CLS_{i+1} such that e_{i+1} follows e_i if any;
4     if there is no pointer been set then
5         return i;
6 return checkLink(n,1,1);

Procedure checkLink(n,SN,SP)
// k represents the size of CLS_SN[n] and s represents the
// size of children(n). Assume that we have a global array
// index[1..s] of pointers to data nodes in CLS_i. index[i]
// represents the position in the CLS_i[n], where the
// bottom of each stack has position 1
1 if SN=s then return −1;
2 Let rs=SN;
3 for each d_i (SP≤i≤k) in CLS_SN[n] do
4     index[SN]=i;
5     Let p be the index of the data node that d_i points to;
6     if p!=null∧(rs=checkLink(n,SN+1,p))<0 then
7         return −1;
8 return rs;
```

The key idea of function checkLink(n,1,1) is to check whether checkLink(n,SN,SP) returns −1 (see procedure checkLink in Algorithm 2), by recursively iterating the data nodes in each stack from bottom to top (line 3) and traversing the stacks in CLS[n] from left to right (line 6). During the computation, the position of the data node of interest in $CLS_{SN}[n]$ is stored in index[SN] (line 4). Notice that, given a stack $CLS_i[n]$ and a data node d_i with element e_i in $CLS_i[n]$, since the pointer of d_i identifies the minimal element e_{i+1} of data node d_{i+1} in the $CLS_{i+1}[n]$ such that e_{i+1} follows e_i, only d_{i+1} and the data nodes above d_{i+1} in $CLS_{i+1}[n]$ are checked (lines 3~7).

Analysis of children linked stacks

Based on CLS, we deduce the following lemmas, which are useful for describing our OTP matching algorithms in Section 4.

Lemma 1 Given a branching node n and an element e that matches node type n, all elements in the corresponding $CLS_i[n]$ of CLS[n] belong to descendants(e). In addition, they match the ith child node n_i

of n, and lie on the same root-leaf path in ascending lexicographical order from bottom to top.

Lemma 2 Given s represents the size of children(n), and index[1..s] is the global array we define in procedure checkLink(n,SN,SP) in Algorithm 2, if and only if createCheckCLS(n,e) returns −1, there is at least a tuple $<e_1,e_2,\ldots,e_s>$ such that each element e_i in the tuple is in $CLS_i[n]$ and isRight(e_i,e_{i+1}) returns true.

Proof The function createCheckCLS(n,e) returns −1 if and only if the function checkLink(n,SN,SP) called by checkLink(n,1,1) returns −1 recursively. When checkLink(n,SN,SP) returns −1 recursively, then a tuple $<e_1,e_2,\ldots,e_s>$ is constructed such that each e_i in the tuple is the index[i]-th element in $CLS_i[n]$ (line 4 in checkLink of Algorithm 2). Suppose d_i is the index[i]-th data node with element e_i in $CLS_i[n]$ and p is the position in $CLS_{i+1}[n]$ to which d_i points. Since p points to the data node d_{i+1} with minimal element e_{i+1} in $CLS_{i+1}[n]$ such that e_{i+1} follows e_i, then the element e_p of the pth data node in $CLS_{i+1}[n]$ follows e_i in CLS_i ($P_{ep}>P_{ei}$). Based on Lemma 1, the data nodes in $CLS_{i+1}[n]$ lie on the same root-leaf path in ascending lexicographical order from bottom to top. Thus, the element e_{i+1} of the index[i+1]-th data node in $CLS_{i+1}[n]$ follows the element e_p of the pth data node in CLS_{i+1} as index[i+1] is the position p or above p ($P_{e(i+1)}\geq P_{ep}$) (lines 3~5). Therefore, given two elements e_i and e_{i+1} in the tuple such that e_i is the element of the index[i]-th data node in $CLS_i[n]$ and e_{i+1} is the element of the index[i+1]-th data node in $CLS_{i+1}[n]$, isRight(e_i,e_{i+1}) returns true ($P_{e(i+1)}\geq P_{ep}>P_{ei}$).

Once the tuple is found, checkLink returns −1 recursively. Otherwise, the index of the child node n_{index} that violates the order requirement is returned (lines 4 and 5 of setAndCheckLink or line 8 of procedure checkLink in Algorithm 2).

Lemma 3 If createCheckCLS(n,e) returns −1, given an element e_a that matches node n and $e_a\in$ancestors(e), the function createCheckCLS(n,e_a) also returns −1.

Proof Based on Lemma 2, if createCheckCLS(n,e) returns −1, there is at least a tuple $T=<e_1,e_2,\ldots,e_s>$ such that each element e_i in the tuple is in CLS_i and isRight(e_i,e_{i+1}) returns true. Since e_a matches node n and $e_a\in$ancestors(e), all the elements in CLS[n] constructed by createCheckCLS(n,e) are also in the CLS[n] created by createCheckCLS(n,e_a). Thus, when createCheckCLS(n,e_a) is called, −1 is returned based on Lemma 2 as the tuple T still exists.

ORDERED TWIG MATCHING ALGORITHMS

In this section, we present our efficient holistic OTP matching algorithm based on CLS, called OTJFast, which needs only to access the labels of the leaf query nodes. Then we propose a new algorithm, named OTJFaster, incorporating three effective optimization rules to skip useless elements.

We will first describe some data structures and notations used by our algorithms, in addition to those introduced in Sections 2 and 3.

Notations and data structures

The main challenge in matching an OTP is that both the P-C or A-D relationships and the order predicates should be satisfied by the elements in the XML document. Thus, we introduce the concept OCE proposed by Lu *et al*.(2005b), which is important in determining whether an element is likely to be involved in an ordered twig query.

Definition 1 (Ordered children extension, OCE) (Lu *et al*., 2005b) Given an OTP Q and an XML document D, we say that an element e_n, which matches a node n such that $n\in Q$ in D, has an OCE, if the following two properties are both satisfied: (1) for each child $n_i\in$children(n), there is an element e_{ni} in D such that e_{ni} is a descendant of e_n and e_{ni} also has an OCE; (2) for each child n_i and r_i=rightSibling(n_i) (if any), there are two elements e_{ni} and e_{ri} such that e_{ni} matches n_i, e_{ri} matches r_i, both e_{ni} and e_{ri} have an OCE, and isRight (e_{ni},e_{ri}) returns true.

The OCE guarantees both the structure relationships and the order requirement of the queries. For example, given the ordered query Q3 in Fig.3b for the XML document in Fig.2a, b1 has an OCE as b1 has descendants e1 and d1 which appear in the correct order according to the query, while b3 does not have an OCE.

Our algorithms keep two data structures for each branching node b: a normal stack S_b and a CLS[b]. Each element cached in stack S_b likely contributes to the solution for the whole OPT. At every point during the computation, the elements in CLS[b] or S_b lie on the same root-leaf path in ascending lexicographical

order from bottom to top. Initially, S_b and CLS[b] are both empty.

The OTJFast algorithm

Now we briefly introduce our algorithm OTJFast (Algorithm 3), which efficiently processes an OTP matching in two phases. In the first phase (lines 1~7), it computes some intermediate solutions to individual root-leaf path patterns and the order requirements of the query pattern are ensured. In the second phase (line 8), these solutions are merged to form the final answers. Note that, although OTJFast shares similarity with the TJFast, it makes important extensions to handle OTP matching based on OCE and CLS.

Algorithm 3 OTJFast(root)

```
1 for each f∈leafNodes(root) do
2     locateMatchedLabel(f);
3 while ∃f∈leafNodes(root): ¬eof(T_f) do
4     f_act=getNext(topBranchingNode);
5     outputSolutions(f_act);
6     advance(T_fact);
7     locateMatchedLabel(T_fact);
8 mergeAllPathSolution();

Procedure locateMatchedLabel(f)
1 while ¬Pcurrent(T_f) matches P_f do
2     advance(T_f);

Function MB(c, b)
1 if isBranching(c) then Let e=max{p|p∈S_c};
2 else Let e=current(T_c);
3 return the set of ancestors, a of e such that b
    can be mapped to a in some mapping of P_c to P_e;
```

Due to the ancestor name vision property of the extended Dewey label, it is easy to test whether an element's path matches the individual root-leaf path pattern. In the optimal case, we output only the path solution that contributes to the solution for the whole OTP. A path solution is useful only if it is merge-joinable to other root-leaf path solutions. As shown in TJFast, if two path solutions can be merged, the necessary condition is that they have the common elements to match the branching query node. Therefore we try to find the likely elements that match branching node n and store them in the corresponding stack S_n (Lu *et al*., 2005a). Moreover, to match an OTP, each common element stored in the normal stacks should also has an OCE.

Function getNext (shown in Algorithm 4) is the core function that is repeatedly called in OTJFast, where we complete two tasks: (1) update the stack S_n of the branching node n with the elements that have an OCE; (2) identify the next stream of a query leaf node to process.

Algorithm 4 getNext(n)

```
1  if isLeaf(n) then
2     return n;
3  else
4     for all n_i∈dbl(n) do
5        f_i=getNext(n_i);
6        if isBranching(n_i)∧empty(S_ni) then
7           return f_i;
8        e_i=max{p|p∈MB(n_i,n)};
9     min=minarg_i{e_i};
10    max=maxarg_i{e_i};
11    for each n_i∈dbl(n) do
12       if ∀e∈MB(n_i,n):e∉ancestors(e_max) then
13          return f_i;
14    Let index=0;
15    for each e∈MB(n_min,n) do
16       if index<0∨(e∈ancestors(e_max)∧(index=
          createCheckCLS(n,e)<0)) then
17          updateSet(S_n,e);
18    return index<0?f_max:f_index;

Procedure updateSet(S,e)
1 Delete any element a in the stack S such that
    a∉ancestors(e) and a∉descendants(e);
2 Add e to stack S;
```

For the first task, we update S_n. Lines 4~14 check the condition (1) of OCE. In particular, before an element e is inserted into the stack S_n, the condition (2) of OCE is checked by the return value of createCheckCLS(n,e) according to Lemma 2 (line 16). Thus, each element cached in the normal stacks has an OCE and is likely to participate in final answers. That is an important extension made by our algorithm OTJFast, compared with TJFast. Note that if there is already some element e_d (e_d∈descendants(e)) such that createCheckCLS(n,e_d) returns −1, we just update the stack S_n with element e according to Lemma 3.

For the second task, the difference between OTJFast and TJFast is that if there is an element e matching node n that has an OCE (line 16), we return $f_{\min}$; otherwise, the leaf node f_{index} that violates the query sibling order is returned (line 18).

Unfortunately, OTJFast may still generate some useless intermediate paths. The reason is that we

check only the common elements for branching nodes by creating and checking the CLS. Once an element e that matches a branching node n is stored in the stack S_n, all the following elements in T_f such that $f\in$ leafNodes(n), which are the descendant elements of e, will be output.

Example 5 Consider the OTP Q3 in Fig.3b on the XML document set visualized in Fig.2a. Initially, current(T_e)=e1 and current(T_d)=d1. The first call of getNext(b) recursively calls getNext(e) and getNext(d) (for e, $d\in$dbl(b) in Q3). Since e and d are leaf nodes, getNext(e)=e and getNext(d)=d. Notice that MB(e,b)={b1} and MB(d,b)={b1}, and createCheckCLS(b,b1) returns −1; thus, b1 is inserted into the stack S_b (line 17 in Algorithm 4). Then getNext(b) returns node d and the path a1/c1/b1/d1 will be output (line 5 in Algorithm 3). Similarly, the second call of getNext(b) returns e and outputs path a1/c1/b1/e1. However, the third call of getNext(b) also returns e and the path a1/c1/b1/e2 will be output, which is useless for the final answers.

Processing the following-sibling relationship and wildcard '*'

It is easy for our algorithm to support wildcard '*' because of the powerful encoding scheme extended Dewey. When OTP contains wildcard '*', the queries can be processed by the algorithms on efficient string matching with 'do not care' symbols (Printer, 1985; Lu *et al.*, 2005a). To match OTP with following-sibling axes, the following-sibling axes are transformed into following axes in the first phase in OTJFast, and then we check the following-sibling relationships when merging the intermediate solutions to form the final answers.

The OTJFaster algorithm

Although OTJFast needs only to access the labels of the leaf query nodes, it still involves some unnecessary computations and the CPU cost could be further improved. In particular, the advantage of skipping elements to save the I/O cost by using traditional indices (such as B^+-tree) is not explored.

In this subsection, we will first describe three optimization rules for OTJFast, and then present our new algorithm OTJFaster.

1. Three optimization rules

Given a subtree Q_n rooted with branching node n, suppose that the normal stack S_n of n is empty, which means that currently there is no matching for Q_n. There are three optimization rules that would enable some unnecessary computations to be avoided in OTJFast.

Rule 1 (Stream null optimization) In case there is any stream T_f of a leaf node $f\in$leafNodes(n) that comes to the end, consider that an arbitrary leaf node $f'\in$leafNodes(n) other than f, current($T_{f'}$) and all elements following current($T_{f'}$) will not participate in any matching of Q_n with a null value of stream T_f. Hence, it is safe to advance $T_{f'}$ to the end.

Rule 2 (Mismatch optimization) Consider two arbitrary leaves f and f' in leafNodes(n). dbl_f and $\text{dbl}_{f'}$ are the two branching nodes in dbl(n). Let e_f be the maximum element in S_{dblf}. MB(dbl_f,n) is defined as the set of all ancestors, a, of e_f such that a can match node n in the path solution of e_f to P_{dblf}. Suppose e_{fmin} is the minimum element in MB(dbl_f,n). If current($T_{f'}$) has a smaller label than that of e_{fmin}, it means that it cannot participate in any matching of Q_n involving current(T_f) based on the definition of e_{fmin}, so the stream $T_{f'}$ can be safely advanced to the first element e such that $P_e\geq P_{\text{efmin}}$. In particular, supposing $f_{\max}$ is the leaf node with the maximum current(T_f), the stream of each other node $f'\in$leafNodes(n) other than $f_{\max}$ can be advanced to e_{fmaxmin}.

Rule 3 (Order optimization) Suppose there is a solution extension of n for unordered twig joins, but n does not have an OCE for ordered twig joins, and 'index' is the first child node that violates the ordered query. It means there is no element in $\text{CLS}_{\text{index}}$ that follows the elements in $\text{CLS}_{\text{index}-1}$. Let a be the highest ancestor of $\text{CLS}_{\text{index}}(n)$. All elements that match n_i (index$<i\leq$size of children(n)) and precede a are out of order too, and are useless and can be pruned.

2. Ordered twig joins faster

In this subsection, we propose our new algorithm, OTJFaster (ordered twig join faster). The main difference between OTJFaster and OTJFast is that OTJFaster makes important extensions to skip the elements that will surely not contribute to final answers according to the above three optimization rules.

To skip the useless elements, we associate each stream T_f of a leaf node f with a new function T_f.fwdBeyond(e), which forwards the T_f to the first element e_f such that $P_{ef}\geq P_e$ (if the value of e is eof, we just forward the stream T_f to the end). We call the

OTJFast algorithm OTJFaster if it calls the new function getNextExt(n) in Algorithm 5, rather than getNext(n).

Algorithm 5 getNextExt(n)

```
1  if isLeaf(n) then
2      return n;
3  else
4      skipElement(n);
5      for all n_i∈dbl(n) do
6          f_i=getNextExt(n_i);
7          if isBranching(n_i)∧empty(S_ni) then
8              return f_i;
9          e_i=max{p|p∈MB(n_i,n)};
10     min=minarg_i{e_i};
11     max=maxarg_i{e_i};
12     Delete any element a in the set S_n such that
           P_a<P_min{p|p∈MB(n_min,n)};
13     for each n_i∈dbl(n) do
14         if ∀e∈MB(n_i,n):e∉ancestors(e_max) then
15             return f_i;
16     Let index=0;
17     for each e∈MB(n_min,n) do
18         if index<0∨(e∈ancestors(e_max)
               ∧(index=createCheckCLS(n,e)<0)) then
19             updateSet(S_n,e);
20     if index>0 then skipElement(n,index);
21     return index<0?f_min:f_index;
```

In line 4 of Algorithm 5, when skipElement is called without the index parameter, the part from line 1 to line 10 of Algorithm 6 is executed, where we deal with two cases. Firstly, according to Rule 1, once any stream T_f of a leaf node f∈leafNodes(n) comes to the end (line 2), the stream $T_{f'}$ of any other arbitrary leaf node f'∈leafNodes(n) can be advanced to the end (lines 3~5). Otherwise, based on Rule 2, supposing that $f_{\max}$ is the leaf node with the maximum current(T_f) (line 7) and $e_{\max}$ is the minimum element∈MB($n_{\max}$,n) (line 8), the stream of each node f'∈leafNodes(n) other than $f_{\max}$ (line 9) can be advanced to $e_{\max}$ safely (line 10).

Example 6 Consider the ordered twig query Q5 in Fig.5b on the document set visualized in Fig.5a. Initially, current(T_b)=b1 and current(T_c)=c1. In OTJFast, the first n times of callings of getNext(a) always return node b, and current(T_b) is advanced from b1 to bn one by one. There is no solution output until the (n+1)th calling of getNext(a) with solution extension <a2,b(n+1),c1>. However, in OTJFaster, by calling lines 6~10 in Algorithm 6 of the procedure skipElement in the first calling of getNextExt(a), T_b is forwarded to b(n+1) directly, and thus n times of callings of getNext(a) can be saved.

Algorithm 6 skipElement(n,index)

```
1  if ¬empty(S_n) then return;
2  if index is not defined then
3      if ∃f∈leafNodes(n):eof(T_f) then
4          for each node f∈leafNodes(n) do
5              T_f.fwdBeyond(eof);
6      else
7          Let f_max be leaf f∈leafNodes(n) with the
               maximum current(T_f);
8          Let e_max=min(p|p∈MB(n_max,n);
9          for each node f∈leafNodes(n) do
10             T_f.fwdBeyond(e_max);
11 else
12     Let a=min(CLS_index−1[n]);
13     Let k=size of leafNodes(n);
14     for each node f_i∈leafNodes(n) (index<i≤k) then
15         T_f.fwdBeyond(a);
```

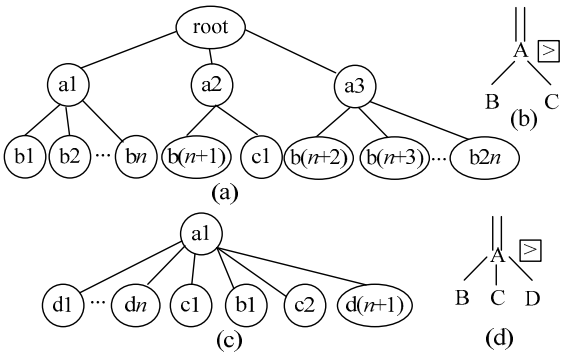

Fig.5 The process for skipping elements
(a) A sample XML document; (b) Q5; (c) A sample XML document; (d) Q6

Example 7 Following Example 6, the second calling of getNextExt(a) returns node c, and current(T_c) is advanced to the end of the stream. In the third calling of getNextExt(a), the two current(T_b)=b(n+2) and current(T_c)=null, and there is an element a2 in the stack S_a. In line 12 of Algorithm 5, we delete the element a2 in the stack. In the fourth calling of getNextExt(a), the procedure skipElement will forward the T_b to the end of the stream according to lines 3~5 of Algorithm 6, and thus the scanning of the elements from b(n+3) to b2n is avoided. Note that line 12 of Algorithm 5 is required; otherwise, element a2 will still be in the stack and the skipElement will return directly in line 1, without skipping the useless calling of getNext.

Based on Rule 3, when there is a solution extension of n if the order checking is ignored and n does not have an OCE for the ordered twig join, function skipElement is called in line 20 of getNextExt (here index is the index of the first child node of children(n) that violates the ordered query). Let a be the highest ancestor of $CLS_{index}[n]$ (line 12). Thus all elements that match n_i (index<i≤size of children(n)) and precede a can be skipped safely (lines 14 and 15).

Example 8 Given the ordered twig query Q6 in Fig.5d on the element set visualized in Fig.5c. Initially, getNextExt(a) returns node c, which is the first child node that violates the ordered query. However, before c is returned, the elements from d1 to dn of node d that are smaller than c1 are skipped by procedure skipElement (lines 11~15 of Algorithm 6). Thus n times of callings of getNext are saved.

Now, we show the correctness of our algorithms. In the function getNext or getNextExt, any element e of a branching node n inserted into the stack S_n has an OCE by constructing and checking the CLS, so the insertion is complete. In addition, in procedure updateSet or line 12 in Algorithm 5, any element e deleted from the stack S_n does not participate in any new solutions, and thus the deletion is safe. In OTJFaster, the function skipElement skips only those elements that will definitely not contribute to the results, based on the three optimization rules. Therefore the skipping is safe too. Thus, the following theorem can be easily established:

Theorem 1 Given an OTP Q and an XML database D, algorithms OTJFast and OTJFaster correctly return all answers for Q on D.

EXPERIMENT

Experimental setup

We used three different datasets in our experiments, one synthetic and two real: (1) XMark (http://monetdb.cwi.nl/xml) is synthetic and is generated by an XML data generator with scale factor=1; (2) DBLP (http://www.informatik.uni-trier.de/~ley/db/) is a shallow and wide document; (3) TreeBank (http://www.cs.washington.edu/research/xmldatasets/) has a very deep recursive structure. Table 1 summarises the characteristics of the three datasets.

Table 1 Characteristics of XML datasets

Parameter	Value		
	DB	XM	TB
Data size (MB)	376.3	113.7	84.1
Number of nodes ($\times 10^6$)	8.80	1.67	2.44
Max./Avg. depth	6/2.9	12/5.5	36/7.9
Region encoding size (MB)	94.8	16.1	24.6
Extended region encoding size (MB)	154.2	25.1	39.4
Extended Dewey size (MB)	71.1	15.1	31.1

DB: DBLP; XM: XMark; TB: TreeBank

We implemented seven OTP matching algorithms: PRIX, LBHJ, OrderedTJ, straightforward-TJFast (STJFast), OTJFast, OTJFaster and OTJFaster-WithIndex. PRIX, LBHJ and OrderedTJ are the three holistic twig join algorithms that attempt the problem of OTP matching. STJFast uses a straightforward post-processing approach, which first processes the query as an unordered twig pattern based on TJFast, and then merges all intermediate path solutions to obtain the answers for an ordered twig joins. OTJFast, OTJFaster and OTJFasterWithIndex are the three algorithms we propose, which need only to access the labels of the leaf query nodes. The algorithms OTJFaster and OTJFasterWithIndex incorporating three optimization rules based on OTJFast both try to avoid the unnecessary computations. The difference is that once there is an element e to skip to, OTJFaster calls the function advance(T_f) to reach e step by step, while OTJFasterWithIndex skips to the element e using B^+-tree index.

Note that OrderedTJ uses a region encoding scheme, PRIX is based on Prűfer's sequence, LBHJ is based on extended Region encoding, and the other four algorithms use extended Dewey. Table 1 also shows the size of different encoding schemes on the three datasets. In particular, the extended Dewey labels were compressed by UTF-8 encoding, as proposed by Tatarinov *et al*.(2002).

Nine ordered twig queries were selected (Table 2), with different structures and combinations of containment or ordered relationships on these three datasets. In particular, TQ3, TQ6, TQ9 contained P-C relationships, preceding-sibling and following-sibling axes, while other queries contained only A-D relationships, preceding and following axes.

Table 2 Ordered twig queries on datasets

Query	Data	Twig queries
TQ1	DB	//inproceedings//sup/following::i
TQ2	DB	//article//sup/proceding::sub
TQ3	DB	//title/sub/proceding-sibling::sup
TQ4	XM	//text//emph[//bold]/following::keyword
TQ5	XM	//item//text[/emp]/following::parlist
TQ6	XM	//keyword/emph/following-sibling::bold
TQ7	TB	//PP//NP[/PNP]/following::VP
TQ8	TB	//NN//VP/proceding::NP
TQ9	TB	//VP/PP/following-sibling::NP

DB: DBLP; XM: XMark; TB: TreeBank

All our experiments were conducted on a PC with a Pentium IV 1.7 GHz processor and 2 GB main memory, running Windows XP. We used the Berkeley-db (http://www.oracle.com/database/berkeley-db/index.html) to implement the B^+-tree index. Taking into account the memory cache, each twig query was executed 20 times and the average results are given.

Experimental results

We show the number of intermediate path solutions output by the different algorithms in Table 3 (OTJFaster and OTJFasterWithIndex had the same number of path solutions as OTJFast, and thus are not included).

Table 3 Number of intermediate path solutions

Query	PR	LB	OT	STJFast	OTJFast	Useful
TQ1	0	*	42	228	97	42
TQ2	0	*	243	548	386	243
TQ3	288	288	288	556	394	288
TQ4	0	*	1968	4869	1986	1968
TQ5	0	*	3697	33922	8457	3697
TQ6	261	261	261	503	265	261
TQ7	0	*	44	110	58	44
TQ8	5	*	16	45	45	16
TQ9	812	812	812	19146	911	812

PR: PRIX; LB: LBHJ; OT: OrderedTJ. * identifies queries not supported by LBHJ because they contain following or preceding axes

OTJFast vs. PRIX We first compared the performance between OTJFast and PRIX under two scenarios, namely kinds of queries supported and the execution time. From Fig.6a we can see that OTJFast performed significantly better than PRIX.

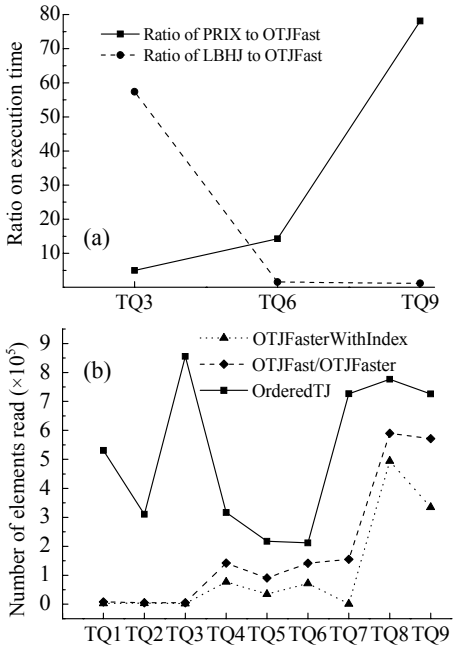

Fig.6 (a) Ratios on execution time of PRIX to OTJFast and LBHJ to OTJFast; (b) The number of elements read

1. Kinds of queries supported. Table 3 shows that there are at least 16 useful results for different queries. But for the queries TQ1, TQ2, TQ4, TQ5 and TQ7 with A-D relationships in the branch edges, the number of path solutions output by PRIX was zero, as PRIX can work correctly only when the query is the subgraph of the XML document. While OTJFast can match all the queries with combinations of any containment relationships and order axes.

2. Execution time. Fig.6a shows that OTJFast outperformed PRIX for all the three queries supported by the algorithms under the scenario execution time. For example, for TQ6, OTJFast needed only 1.3 s, while PRIX took 20 s (almost 15 times as long). This is because OTJFast needs to scan only the elements of query leaf nodes, while PRIX processes the subsequence matching by scanning the elements of all query nodes one by one. In particular, when the document has a recursive deep structure, the gap between OTJFast and PRIX becomes more apparent. For example, for TQ9 on the dataset TreeBank, OTJFast needed only 6.5 s, but PRIX took 510 s (nearly 80 times as long).

OTJFast vs. LBHJ From Table 3 and Fig.6a, OTJFast also outperformed LBHJ. LBHJ supports only two order axes: following-sibling and preceding-sibling (Table 3). In our experiment, only three of the

nine queries (TQ3, TQ6, TQ9) could be processed by LBHJ, while OTJFast could match the queries with combinations of any containment relationships and order axes. OTJFast also used less execution time than LBHJ for the three queries (Fig.6a). For example, for TQ6, LBHJ took 7.8 s, but OTJFast needed only 0.14 s (over an order of magnitude faster). This can be explained by the fact that OTJFast reduces the I/O cost of LBHJ by reading only labels of query leaf nodes.

OTJFast vs. OrderedTJ OTJFast shares similarity with OrderedTJ; however, the biggest difference between the two algorithms is that OrderedTJ needs to scan the labels of all query nodes, while OTJFast needs only to access the elements of query leaf nodes. OTJFast outperformed OrderedTJ for all nine queries in terms of the number of elements scanned, the calling times of the core function getNext, size of disk files scanned and the total execution time (Figs.6b and 7). Fig.6b shows that OTJFast read far fewer elements than OrderedTJ. For example, for TQ2, OrderedTJ read 310472 elements, but OTJFast read only 5170 (nearly two orders of magnitude fewer). Fig.7b shows a similar trend in terms of the size of disk file scanned. OTJFast used much less execution time and called the getNext function far fewer times than OrderedTJ (Figs.7a and 7c).

The weakness of OTJFast is that it generates more intermediate results compared with OrderedTJ, which is I/O optimal when the query contains only A-D relationships from the 2nd branching edge. Generally, the number of intermediate path solutions of OTJFast was larger than that of OrderedTJ (Table 3). For example, for TQ1, OrderedTJ generated 42 intermediate path solutions, while OTJFast generated 97.

OTJFast vs. STJFast Since both need to access only the elements of the query leaf nodes, the number of elements read, the size of disk files scanned and the calling times of the getNext function of OTJFast and STJFast are quantitatively the same. However, in contrast to STJFast, OTJFast caches only the elements of the branching node in the stack that are guaranteed to contribute to the final results for ordered twig queries based on the CLS. Thus, OTJFast produces fewer useless intermediate solutions. For example, for TQ5, the number of useless intermediate solutions using OTJFast was only 8457, compared with 33922 for STJFast (about 3 times greater). OTJFast greatly reduces the number of intermediate results, thus saving CPU cost (Figs.8a and 9a). However, when the number of intermediate results in OTJFast is almost the same as for STJFast, OTJFast will consume more time than STJFast (Fig.10a), since OTJFast needs to create and check the CLS during the query execution.

OTJFaster vs. OTJFast Figs.8~10 show that our new algorithm OTJFaster outperformed or at least performed the same as OTJFast for all queries. In terms of execution time, OTJFaster was up to 3%~15% faster than OTJFast. In addition, the numbers of calling times of the core function for OTJFaster were significantly fewer than those for OTJFast. Note that the core function of OTJFaster or OTJFasterWithIndex is getNextExt instead of getNext. For example, in Fig.10c, for TQ7, OTJFast called getNext 66426 times, but OTJFaster called getNextExt only 759 times (two orders of magnitude fewer). This is because we identify and skip the computations of the useless elements in OTJFaster. However, OTJFaster avoids only the unnecessary computations and still needs to access the useless elements one by one. Thus, the number of elements read, the size of the intermediate path solutions and the size of disk files scanned using OTJFaster are as the same as those for OTJFast.

OTJFasterWithIndex vs. OTJFast Figs.8~10 show that OTJFasterWithIndex outperformed OTJFast for all queries. For instance, given query TQ7 in terms of execution time, OTJFasterWithIndex was up to 60 times faster than OTJFast (27 ms vs. 1632 ms). A similar trend was found in other examples and in terms of other metrics. In addition, the advantage of OTJFasterWithIndex over OTJFast was even greater for the DBLP dataset on TQ1, TQ2 and TQ3. In particular, for TQ1 in terms of the number of elements read, OTJFasterWithIndex scanned fewer than 1/3 of the elements scanned by OTJFast (2781 vs. 7986). Meanwhile, in terms of the number of calling times of the core function, OTJFast called twice as many times as OTJFasterWithIndex. The performance advantage of OTJFasterWithIndex over OTJFast arises mainly from the fact that OTJFasterWithIndex reduces the number of function calls and makes use of available indexes such as B^+-tree to skip the accessing of useless elements.

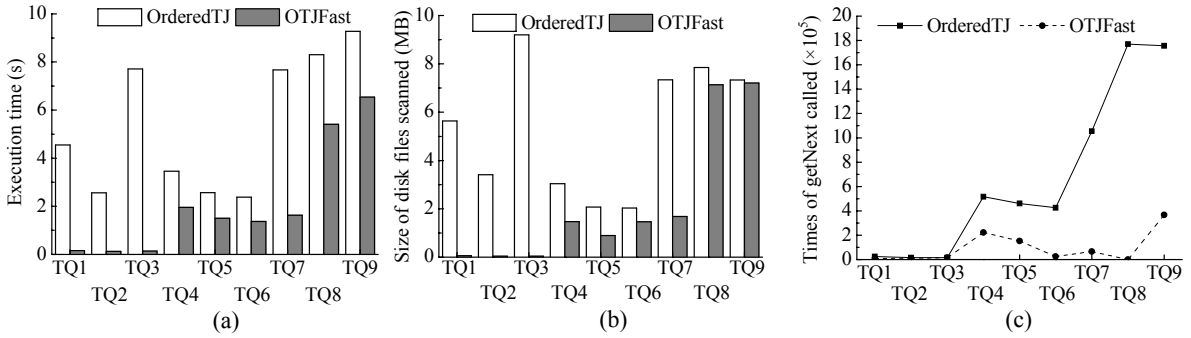

Fig.7 OTJFast versus OrderedTJ

(a) Execution time; (b) Size of disk files scanned; (c) Number of times getNext called

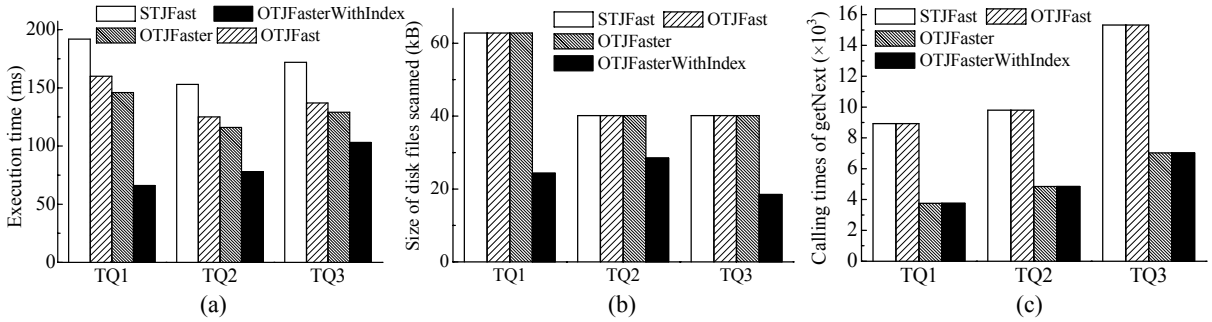

Fig.8 STJFast, OTJFast, OTJFaster and OTJFasterWithIndex on DBLP data

(a) Execution time; (b) Size of disk files scanned; (c) Number of times getNext called

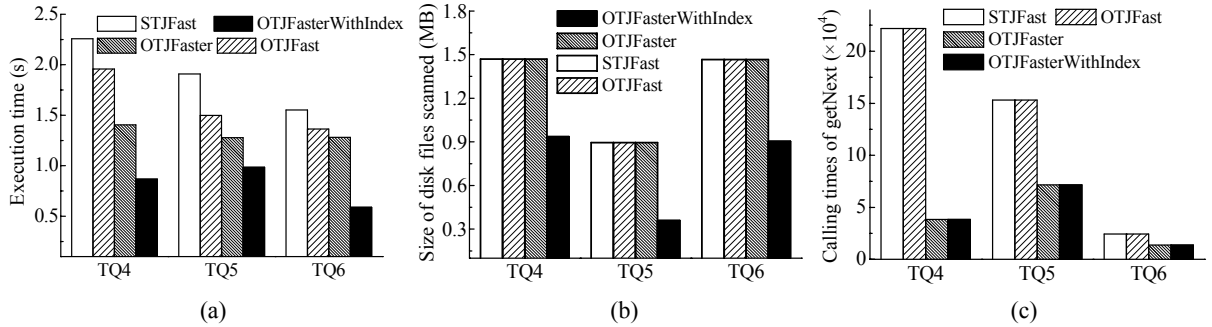

Fig.9 STJFast, OTJFast, OTJFaster and OTJFasterWithIndex on XMark data

(a) Execution time; (b) Size of disk files scanned; (c) Number of times getNext called

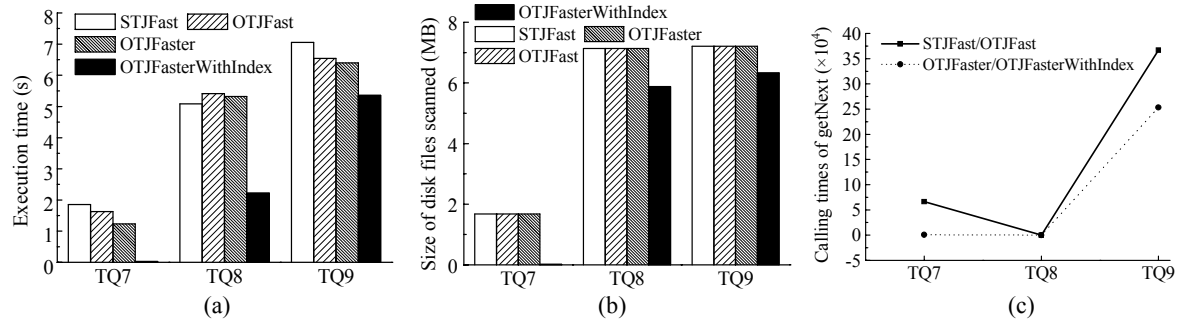

Fig.10 STJFast, OTJFast, OTJFaster and OTJFasterWithIndex on TreeBank data

(a) Execution time; (b) Size of disk files scanned; (c) Number of times getNext called

RELATED WORKS

The state-of-the-art algorithms process XML twig joins in a holistic way. The first holistic twig join algorithm, namely TwigStack, was proposed by Bruno *et al*.(2002). Jiang HF *et al*.(2003) proposed a general algorithm, named TSGeneric$^+$, to address the problem of efficient processing twig joins on indexed XML documents. To overcome the small class of optimal queries in TwigStack, Lu *et al*.(2004) proposed a novel algorithm, named TwigStackList, which is I/O optimal for queries with only ancestor-descendant relationships below branching nodes using a look-ahead approach. To explore the potential benefit to CPU cost, an efficient holistic twig join algorithm, called TJEssential (Li *et al*., 2007), was proposed incorporating three optimization rules, which can avoid self-nested, order and null-stream suboptimality. Chen *et al*.(2006) proposed the first GTP matching solution based on a hierarchical stack encoding scheme, called twig^{2}stack, to handle queries with repeated labels. However, it reduced the intermediate results at the expense of a huge memory requirement. All above algorithms are based on the region encoding scheme (Zhang *et al*., 2001). To gain more information from the labels, a novel powerful labeling scheme, called extended Dewey, was proposed by Lu *et al*.(2005a). Based on extended Dewey, Lu *et al*.(2005a) proposed a novel algorithm, called TJFast, which needs to access only the elements of the leaf query nodes.

A key issue that most of previous algorithms have ignored is whether the ordered twig queries are supported. Three algorithms for holistic XML OTP matching have been published recently. PRIX (Rao and Moon, 2004; 2006) transforms both XML data and queries into sequences and answers XML queries through subsequence matching which supports OTP matching inherently. OrderedTJ (Lu *et al*., 2005b) based on the region encoding scheme and LBHJ (Zhu *et al*., 2008) based on the extended region encoding scheme both process OPT matching in a holistic way. However, all these three algorithms have poor performance.

CONCLUSION

In this paper, we address the problem of efficient processing of OTP matching based on extended Dewey. We first proposed the children linked stacks (CLS) encoding scheme to represent compactly the partial ordered twig results. Then we designed a novel algorithm based on CLS, called OTJFast, which needs only to access the labels of the query leaf nodes. Furthermore, we proposed a new algorithm, OTJFaster, incorporating three effective optimization rules to avoid unnecessary computations. Hence, the number of elements read can be reduced and a large number of useless recursive calls can be avoided. In particular, OTJFaster works well with available indices such as B$^+$-tree. Experimental results showed the efficiency of our algorithms.

As part of future work, we would like to illustrate the optimality of our ordered twig join algorithms.

References

Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M., Srivastava, D., Wu, Y., 2002. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. Proc. 18th Int. Conf. on Data Engineering, p.141-152. [doi:10.1109/ICDE.2002.994704]

Berglund, A., Boag, S., Chamberlin, D., Fernandez, M.F., Kay, M., Robie, J., Simeon, J., 2002. XML Path Language (XPath) 2.0. Technical Report WD-xpath20-20020816. W3C.

Boag, S., Chamberlin, D., Fernandez, M.F., Florescu, D., Robie, J., Simeon, J., 2002. XQuery 1.0: An XML Query Language. Technical Report WD-xquery-20020816. W3C.

Bruno, N., Koudas, N., Srivastava, D., 2002. Holistic Twig Joins: Optimal XML Pattern Matching. Proc. ACM SIGMOD Int. Conf. on Management of Data, p.310-321. [doi:10.1145/564691.564727]

Chen, S., Li, H.G., Tatemura, J., Hsiung, W.P., Agrawal, D., Candan, K.S., 2006. Twig^{2}Stack: Bottom-up Processing of Generalized-tree-pattern Queries over XML Documents. Proc. 32nd Very Large Database, p.283-294. [doi:10.1145/1321440.1321446]

Chen, T., Lu, J.H., Ling, T.W., 2005. On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques. Proc. ACM SIGMOD Int. Conf. on Management of Data, p.455-466. [doi:10.1145/1066157.1066209]

Jiang, H.F., Wang, W., Lu, H., 2003. Holistic Twig Joins on Indexed XML Documents. Proc. 29th Very Large Database, p.273-284.

Jiang, J.H., Chen, G., Shou, L.D., Chen, K., 2008. OTJFast: Processing Ordered XML Twig Join Fast. Proc. IEEE Asia-Pacific Services Computing Conf., p.1289-1294. [doi:10.1109/APSCC.2008.15]

Jiang, Z.W., Luo, C., Hou, W.C., 2006. An Efficient One-phase Holistic Twig Join Algorithm for XML Data. Proc. 15th ACM Int. Conf. on Information and Knowledge

Management, p.786-787. [doi:10.1145/1183614.1183730]

Li, G.L., Feng, J.H., Zhang,Y., Zhou, L.Z., 2007. Efficient Holistic Twig Joins in Leaf-to-Root Combining with Root-to-Leaf Way. Proc. 12th Int. Conf. on Database Systems for Advanced Applications, p.834-849. [doi:10.1007/978-3-540-71703-4_69]

Lu, J.H., Chen, T., Ling, T.W., 2004. Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach. Proc. 13th ACM SIGMOD Int. Conf. on Management of Data, p.533-542. [doi:10.1145/1031171.1031272]

Lu, J.H., Ling, T.W., Chan, C.Y., Chen, T., 2005a. From Region Encoding to Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. Proc. 31st Very Large Database, p.193-204.

Lu, J.H., Ling, T.W., Yu, T., Li, C., Ni, W., 2005b. Efficient Processing of Ordered XML Twig Pattern. Proc. Int. Conf. on Database and Expert Systems Applications, p.300-309. [doi:10.1007/11546924_30]

Printer, R.Y., 1985. Efficient String Matching with Don't Care Patterns. *In*: Apostolico A., Galil, Z. (Eds.), Combinatorial Algorithms on Words. NATO Advanced Science Institute Series F: Computer and System Sciences. Springer-Verlag, New York, p.11-29.

Prűfer, H., 1918. Neuer beweis eines satzes űber permutationen. *Arch. Math. Phys.*, **27**:142-144 (in Spanish).

Rao, P., Moon, B., 2004. PRIX: Indexing and Querying XML Using Prűfer Sequences. Proc. 20th Int. Conf. on Data Engineering, p.288-299. [doi:10.1109/ICDE.2004.1320005]

Rao, P., Moon, B., 2006. Sequencing XML data and query twig for fast pattern matching. *ACM Trans. Database Syst.*, **31**(1):299-345. [doi:10.1145/1132863.1132871]

Tatarinov, I., Viglas, S.D., Beyer, K., Shanmugasundaram, J., Shekita, E., Zhang, C., 2002. Storing and Querying Ordered XML Using a Relational Database System. Proc. ACM SIGMOD Int. Conf. on Management of Data, p.204-215. [doi:10.1145/564691.564715]

Zhang, C., Naughton, J., Dewitt, D.J., Luo, Q., Lohman, G.M., 2001. On supporting containment queries in relational database management systems. *ACM SIGMOD Rec.*, **30**(2):425-436. [doi:10.1145/376284.375722]

Zhu, J.Q., Wang, W., Meng, X.F., 2008. Efficient Processing of Complex Twig Pattern Matching. Proc. 9th Int. Conf. on Web-Age Information Management, p.135-140. [doi:10.1109/WAIM.2008.54]

Zografoula, V., Nick, K., Divesh, S., Vassilis, J.T., 2005. Answering Order-based Queries over XML Data. Proc. 14th Int. World Wide Web Conf., p.1162-1163. [doi:10.1145/1062745.1062919]