Extended XML Tree Pattern Matching: Theories and Algorithms

Jiaheng Lu, Tok Wang Ling, Zhifeng Bao and Chen Wang

Abstract—As business and enterprises generate and exchange XML data more often, there is an increasing need for efficient processing of queries on XML data. Searching for the occurrences of a tree pattern query in an XML database is a core operation in XML query processing. Prior works demonstrate that holistic twig pattern matching algorithm is an efficient technique to answer an XML tree pattern with parent-child (P-C) and ancestor-descendant (A-D) relationships, as it can effectively control the size of intermediate results during query processing. However, XML query languages (e.g. XPath, XQuery) define more axes and functions such as negation function, order-based axis and wildcards. In this article, we research a large set of XML tree pattern, called *extended XML tree pattern*, which may include P-C, A-D relationships, negation functions, wildcards and order restriction. We establish a theoretical framework about "*matching cross*" which demonstrates the intrinsic reason in the proof of optimality on holistic algorithms. Based on our theorems, we propose a set of novel algorithms to efficiently process three categories of extended XML tree patterns. A set of experimental results on both real-life and synthetic data sets demonstrate the effectiveness and efficiency of our proposed theories and algorithms.

Index Terms—Query processing, XML/XSL/RDF, algorithms, tree pattern

1 Introduction

As business and enterprisers generate and exchange XML data more often, there is an increasing need for efficient processing of queries on XML data. An XML query pattern commonly can be represented as a rooted, labeled tree (or called twig). For example, Figure 1(a) shows an example XPath query: $A[B]/C$ and the corresponding XML tree pattern. This query finds all node C that has the parent A which has another child B. In Figure 1(b), the query answers are nodes "C_1" and "C_2".

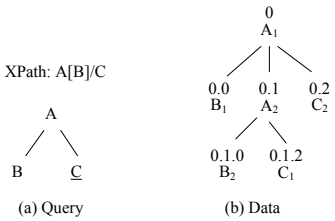


Fig. 1. Example XML tree query and document. "_" denotes the return node in query. The answers are C_1 and C_2. The digital labels will be explained later.

Efficient matching of XML tree patterns has been widely considered as a core operation in XML query processing. In recent years, many methods ([9], [13], [3], [11], [4], [25]) have been proposed to match XML tree queries efficiently. In particular, Khalifa et al. [1] proposed a stack-based algorithm to match binary structural relationship including parent-child (P-C) and ancestor-descendant (A-D) relationships. The limitation of their method is that the size of useless intermediate results may become very large, even if the final results are small. Bruno et al. [3] proposed a novel holistic twig join algorithm named TwigStack, which processes the tree pattern *holistically* without decomposing it into several small binary relationships. TwigStack guarantees that there are no "useless" intermediate results for queries with only *ancestor-descendant* (A-D) relationships. In other words, TwigStack is *optimal* for tree pattern queries with only A-D edges [8]. Many other recent works then examine how to enlarge the optimal query class of holistic algorithms [14], to speedup performance using indexes [11], [5], to devise new data streaming strategies [6], and to propose efficient and dynamic labeling schemes [16]. These algorithms have proven highly promising and make their way into XML query processing applications, in both academic and industrial settings [19]. But we still have the following observations upon the above existing works.

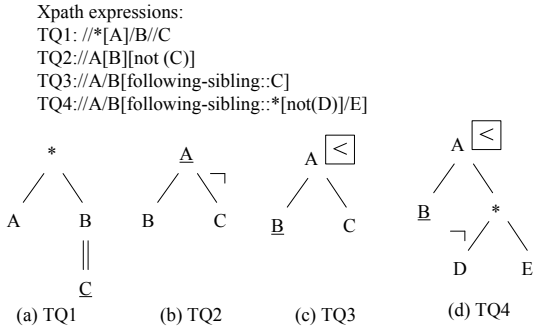


Fig. 2. Example extended XML tree pattern queries. "_" denotes the return node in query

- *Jiaheng Lu is with the School of Information and DEKE, MOE in Renmin University of China. Contact Author. E-mail: jiahenglu@ruc.edu.cn*
- *Tok Wang Ling and Zhifeng Bao are with the School of Computing, National University of Singapore. E-mail: {lingtw, baozhife}@comp.nus.edu.sg*
- *Chen Wang is a Staff Researcher at IBM Research - China. E-mail: wangcwc@cn.ibm.com*

Extended XML tree pattern Previous algorithms focus on XML tree pattern queries with only P-C and A-D relationships. Little work has been done on XML tree queries which may contain wildcards, negation function and order restriction, all of which are frequently used in XML query languages such as XPath and XQuery. In this article, we call an XML tree pattern with negation function, wildcards and/or order restriction as *extended XML tree pattern*. Figure 2, for example, shows four extended XML tree patterns. Query (a) includes a wildcard node "*", which can match any single node in an XML database. Query (b) includes a negative edge, denoted by '$\neg$'. This query finds A that has a child B, but has *no* child C. In XPath language [2], the semantic of negative edge can be presented with "*not*" boolean function. Query (c) has the order restriction, which is equivalent to an XPath "//A/B[following-sibling::C]". The '$<$' in a box shows that all children under A are ordered. The semantics of order-base tree pattern is captured by a mapping from the pattern nodes to nodes in an XML database such that the *structural* and *ordered* relationships are satisfied. Finally, Query (d) is more complicated, which contains wildcards, negation function and order restriction.

Optimality of holistic algorithms Previous XML tree pattern matching algorithms do not fully exploit the "optimality" of holistic algorithms. TwigStack [3] guarantees that there is no useless intermediate result for queries with only A-D relationships. Therefore, TwigStack is *optimal* for queries with only A-D edges. Another algorithm TwigStackList [14] enlarges the optimal query class of TwigStack by including P-C relationships in non-branching edges. A natural question is whether the optimal query class of TwigStackList can be further improved. Hence, the current open problems include (1) how to identify a larger query class which can be processed optimally and (2) how to efficiently answer a query which cannot be guaranteed to process optimally. Note that earlier works in [8], [21] already showed that no algorithm is optimal for queries with any arbitrary combinations of A-D and P-C relationships. This article explores the challenges and shows the promise of a novel theoretical framework called "*matching cross*" to identify a large optimal query class for posing extended XML tree queries.

Return nodes in twig pattern queries In a practical application, only part of query nodes belong to *return* nodes (or called *output* nodes interchangeably). Take the XPath "//A[B]//C" as an example, only C element and its subtree are answers. The current "modus operandi" (e.g. [12], [3], [16]) is that they first find the query answer with the combinations of all query nodes, and then do an appropriate projection on those return nodes. Such a post-processing approach has an obvious disadvantage: it outputs many matching elements of non-return nodes that are unnecessary for the final results. In this article, we develop a new encoding method to record the mapping relationships and avoid outputting non-return nodes.

1.1 Main results

In general, given an *extended XML tree pattern query* which may include P-C, A-D relationships, order restriction, negation function and wildcards, we consider the problem efficiently matching the extended XML tree query. Our algorithm aims at identifying a large queries class which can be optimally processed. Like previous papers on XML tree pattern matching (e.g. [3], [12], [16]), in this article, we call a holistic algorithm "optimal" for a kind of query class, if it guarantees that any output intermediate results contribute to final answers. For example, previous algorithm TwigStack [3] is optimal for query class with only A-D edges, and TwigStackList [14] is optimal for queries with only A-D relationships in branching edges.

We investigate three categories of extended XML tree patterns (See Figure 3): (1) queries with P-C, A-D relationships and wildcards, denoted as $Q^{/,//,*}$; and (2) queries with P-C, A-D relationships, wildcards and order restriction, denoted as $Q^{/,//,*,<}$; and (3) queries with P-C, A-D relationships, wildcards, order restriction and negation function, denoted as $Q^{/,//,*,<,\neg}$. For each category, we identify the respective optimal query class.

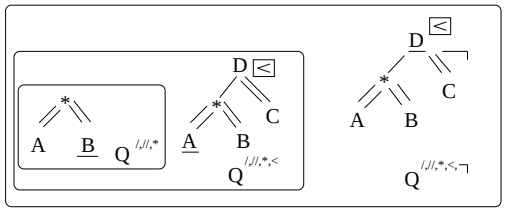


Fig. 3. Three categories of extended XML tree patterns and example optimal queries

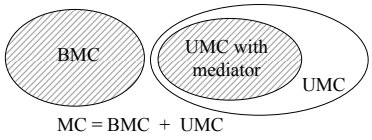


Fig. 4. Illustration to the relationship between BMC and UMC. The shaded portions demonstrate the optimal query classes.

The technical contribution of this article are summarized as follows.

- We build a theoretical framework on optimal processing of XML tree pattern queries. We show that "**matching cross**" is the key reason to result in the sub-optimality of holistic algorithms. Intuitively, matching cross describes a *dilemma* where holistic algorithms have to decide whether to output *useless* intermediate result or to miss *useful* results. The fact that TwigStack[3] is optimal for queries with only A-D relationships can be explained that no *matching cross* can be found for any XML document with respect to queries with A-D edges. We classify *matching cross* to *bound* and *unbounded matching cross* (BMC and UMC. See Figure 4). We develop theorems to show that only part of UMC (i.e. UMC with mediator) can force holistic algorithms to potentially output useless intermediate results.
- Based on the theoretical analysis, we develop a series of holistic algorithms TreeMatch to achieve a large optimal query class for three categories of queries (i.e.

$Q^{/,//,*}$,$Q^{/,//,*,<}$ and $Q^{/,//,*,<,\neg}$). Our main technique is to use a concise encoding to present matching results, which leads to the reduction of useless intermediate results.

- We conducted an extensive set of experiment on synthetic and real data set for performance comparison. We compared TreeMatch with previous four holistic XML tree pattern matching algorithms. The experimental results show that our algorithm can correctly process extended XML tree patterns, achieving performance speedup for tested queries and data sets, even in their restricted focus. The improvement mainly owes to the reduction of the size of intermediate results.

1.2 Outline

The rest of the article is organized as follows. Section 2 gives the preliminaries about research problem and the processing model. Section 3 shows a set of theories about matching cross and Section 4 presents an extended XML tree pattern matching algorithm called TreeMatch. Section 5 presents thorough experimental studies between the novel algorithms and the prior methods. Finally, Section 6 presents previous work related to the XML tree pattern matching and Section 7 concludes this article.

2 PRELIMINARIES

2.1 Modeling of XML data and extended tree pattern query

An XML database D is usually modeled as a rooted, node-labeled tree (in this article, we use D to represent the database and the related tree model exchangeably without specific declaration), element tags and attributes are mapped to nodes in the trees and the edges are used to represent the direct nesting relationships. Our primary focus is on element nodes; and it is not difficult to extend our methods to process the other types of nodes, including attribute and character data. For convenience, we distinguish between query nodes and database nodes by using the term "node" to refer to a query node and the term "element" to refer to a data element in D.

An extended tree query Q describes a complex traversal of the XML document and retrieves relevant tree-structured portions of it. The nodes in Q include element tags, attributes and character data. We use "*" to denote the wildcard, which can match any single tree element. There are four kinds of query edges, which are the four combinations between (*positive*, *negative*) and (*parent-child*, *ancestor-descendant*). For example, in Figure 2 (b), (A, B) is a *positive parent-child* edge and (A, C) is a *negative parent-child* edge. We use a symbol "$\neg$" to denote a negative edge. There are two kinds of query node: *ordered* and *unordered* node. We use "$<$" in a box to denote the *ordered* node, otherwise it is an *unordered* node. For example, the node A in Figure 2 (c) and (d) are ordered nodes. In each extended tree query pattern, there is one or multiple nodes which are assigned as the selected return nodes, denoted with an underline. For example, in Figure 2 (a), C is the selected return node.

Given an extended tree query Q with n selected return nodes and an XML database D, a *match* of Q in D is identified by a mapping from nodes in Q to the elements in D, such that: (i) query node types (i.e. tag name) are satisfied by the corresponding database elements and wild-cards "*" can match any single database element; (ii) the positive edge relationships (including positive parent-child and positives ancestor-descendant edges) between query nodes are satisfied by the corresponding database elements; (iii) the negative edge relationships (including negative parent-child and negative ancestor-descendant edges) are satisfied, that is, no corresponding database element pairs exist; and (iv) the order relationship of children of each ordered node is satisfied by the corresponding database elements. The answers of a query can be represented as a set of database elements, where each element identifies a distinct match of the selected *return* nodes on D. For example, Fig 5 shows an example mapping relationship between an extended XML tree pattern and a document tree.

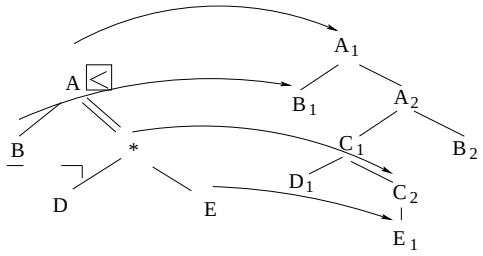


Fig. 5. Mapping relationship between an extended tree pattern and a document tree

2.2 Labeling Schemes

Most XML query processing algorithms on XML documents rely on certain labeling schemes, such as region encoding scheme [27], prefix scheme [13], ORDPATH [19], extended Dewey scheme [16]. In this article, we use the *extended Dewey* labeling scheme, proposed in paper [16], to assign each node in XML documents a sequence of integers to capture the structure information of documents.

Extended Dewey labeling scheme is a variant scheme of the prefix labeling scheme. In the *prefix* labeling scheme, the *root* is labeled by an empty string and for a non-root element u, $label(u) = label(v).n$, where u is the n-th child of v. In *Extended Dewey* labeling scheme, each label provides complete information about ancestors' *names* and *labels*. For example, given an element e with label "1.2.3", *prefix* labeling schemes can tell us parent(e)="1.2" and grandparent(e)="1", but *extended Dewey* labeling scheme can also tell us the tag name of elements, say, tag(e)='A', tag(parent(e))='B' and tag(grandparent(e))='C'. In order to achieve this goal, paper [16] uses module function to encode the element tag information to prefix labels, and use finite state transducer (FST) to decode the the types information for a single *extended Dewey* label. The details of modular function and FST are out of scope of this article. But for the purpose of understanding this article here, readers only need to know that in the *extended*

Dewey labeling scheme, from the label of a single element, we can derive all the elements names along the path from the root to the element. And the complete path information in *extended Dewey* labels enables holistic algorithms to scan only leaf query nodes to answer an XML query.

2.3 Basic properties of algorithms

The algorithms for XML tree pattern matching proposed in this article have two basic yet important properties, as follows.

Single-direction scan We adopt a structure, named label list, associated with each query node. The label list is a posting list (or inverted list) containing the *extended Dewey* labels of XML elements which have the same name, and all elements are ordered according to the *document order*. We use T_A to denote the label list for query node A. There is a cursor for each list. It moves in the single direction to scan all elements once in increasing order. Each label in a list can be read only once.

Bounded main memory For a large class of queries, the main memory requirement of our algorithm is linear to the number of nodes in the longest path of XML database, which is usually small. Therefore, our solution would be scalable to a very large document with a small main memory requirement.

Recall that the existing algorithms such as `TwigStack` [3], `TwigStackList` [14], `TJFast` [16] also have the first property. That is, they keep the single-direction scan of the document. But for the second property, those algorithms guarantee the bounded main memory for a small class of queries. This article makes the contribution to propose algorithms to achieve this property for a much larger class of queries with negation predicates, wildcards and order restriction.

3 Theoretical Analysis

In this section, we establish a theoretical framework about "*matching cross*" which demonstrates the intrinsic reason for the sub-optimality of existing holistic algorithms. The purposes of our study are (i) to provide insight into the characteristics of the holistic algorithms, and thus promotes our understanding about their behaviors; and (ii) to lead to novel algorithms that can guarantee a larger optimal query class and realize better query performance.

3.1 Matching Cross

The existing holistic algorithms ([11], [16]) consist of two phases: (i) in the first phase, a list of path solutions is output as intermediate path solutions and each solution matches the individual root-to-leaf path expression; and (ii) in the second phase, the path solutions are merged to produce the final answers for the whole twig query. However, for queries with *parent-child* (P-C) relationships, the state-of-the-art algorithms cannot guarantee that each intermediate solution output in the first phase is useful to merge in the second phase. In other words, many useless intermediate results (i.e. path solutions) may be produced in the first phase, which is called the *sub-optimality* of algorithms, as further illustrated in the following example.

Example 1: Consider the document and query in Figure 1 again. First, A_1, B_1 and C_1 are scanned. Although B_1 has the parent A_1, *at this point*, we do not know whether A_1 has a child C. Now holistic algorithms meet a dilemma, that is, whether to output possibly "*useless*" intermediate path (A_1, B_1), or to miss the potential correct answer related to A_1. (This dilemma is formalized as "matching cross" later.) In order to guarantee the completeness of query answers, previous methods (e.g. `TwigStack`) directly output the path (A_1, B_1), which may become "*useless*" intermediate path solution if there were no C_2 in data. □

We generalize the observation in Example 1 into a concept, called **matching cross**. Before proceeding, we need a preliminary definition called *first match*.

Definition 3.1: (**First Match**) *Given an XML database D and a query Q, assume that A, B are two query nodes in Q. Let A_i be an element in the label list T_A. We say that B_j in T_B is the first match of A_i, denoted as $FM(A_i,B)=B_j$, if and only if (A_i, B_j) appears in a match binding to query Q and there is no other element B_k, $k < j$ such that (A_i, B_k) is also in a match binding.*

Note that in the above definition, all elements' labels in T_A and T_B are sorted by document order; and thus B_k is a preceding element of B_j as $k < j$. For example, in Figure 1(b), FM(B_1,C)=C_2 and FM(B_2,C)=C_1. In addition, note that $FM(A_i,B)=B_j$ does not guarantee $FM(B_j,A)=A_i$.

Definition 3.2: (**Matching Cross**) *Given an XML database D and a query Q, assume that A, B are two query nodes in Q. Let A_i, A_j ($i < j$) be two elements in label list T_A; and $B_{i'}$, $B_{j'}$ ($i' < j'$) be two elements in T_B. We say that the 4-tuple $<A_i, A_j, B_{i'}, B_{j'}>$ is a* matching cross *on D with respect to Q if and only if $FM(A_i,B)=B_{j'}$ and $FM(B_{i'},A)=A_j$* (See Fig. 6) □

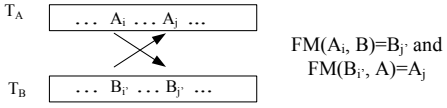


Fig. 6. Illustration to matching cross

It is easy to prove that if $<A_i, A_j, B_{i'}, B_{j'}>$ is matching cross, then $<B_{i'}, B_{j'}, A_i, A_j>$ is also a matching cross.

In Figure 1(b), $<B_1, B_2, C_1, C_2>$ is a matching cross since the first match of B_1 is C_2, and that of C_1 is B_2. Note that B_1 and C_1 are not in the same match binding. The existence of *matching cross* forces holistic algorithms to output *uncertain* intermediate path solutions and may cause their *sub-optimality*.

The following lemma identifies a query class, with respect to which we cannot find any document with matching cross.

Lemma 1: Suppose Q is a tree pattern query with only ancestor-descendant (A-D) relationships in all edges, given any document D, there is no matching cross on D with respect to Q.

PROOF: We prove it by contradiction. Assume that a matching cross $<A_i, A_j, B_{i'}, B_{j'}>$ occurs when evaluating Q on document D. Let "$\prec$" denote preceding relationship in document order. Then $A_i \prec A_j$ and $B_{i'} \prec B_{j'}$. There are the following two cases.

(1) A and B appear in the same path in the query Q. Without loss of generality, assume A is ancestor of B in Q. There are two sub-cases: case(1.1) A_i is an ancestor of A_j in document D. Since (A_j,$B_{i'}$) is a match binding, A_j is an ancestor of $B_{i'}$. Since all edges in query are A-D relationships, (A_i,$B_{i'}$) is also a match binding, which contradicts that A_j is the first match of $B_{i'}$. Case(1.2) A_i and A_j are in different data paths. Since (A_j,$B_{i'}$) is a match binding, A_j is an ancestor of $B_{i'}$. So $A_i \prec B_{i'}$ and $B_{i'}$ is not in the same data path with A_i. Since $B_{i'} \prec B_{j'}$, $B_{j'}$ is also not in the same data path with A_i, which contradicts that $B_{j'}$ is the first match of A_i.
(2) Assume that A and B are in the different root-to-leaf pathes in Q. Assume that node C is the lowest common ancestor of A and B in Q. Then there are two matching bindings $(A_i, B_{j'}, C_1)$ and $(A_j, B_{i'}, C_2)$. Consider two sub-cases. (2.1): C_1=C_2, then it is easy to see that $<A_i$,$B_{i'}>$ is also a matching binding, which contradicts $B_{j'}$ is the first match of A_i. Case (2.2): $C_1 \neq C_2$. Without the loss of generality, assume that $C_1 \prec C_2$, then C_1 is an ancestor of C_2, otherwise there is no overlap in C_1, C_2 subtrees, which contracts that $B_{i'} \prec B_{j'}$. Then (A_i, $B_{i'}$, C_2) is also a matching binding, which contracts that $B_{j'}$ is the first match of A_i. □

According to Lemma 1, no matching cross can occur during evaluating queries with only A-D relationships. This lemma shows the intrinsic reason why the previous algorithm TwigStack [3] can guarantee the optimality for queries with only A-D relationships, as there is no matching cross in such cases. But note that an existing algorithm TwigStackList [14] can identify a larger query class to guarantee the optimality than that of TwigStack. This fact implies that a certain kind of *matching cross* does not necessarily cause the *sub-optimality* of holistic algorithms, as illustrated follows.

Definition 3.3: (**bounded matching cross**) *Given a query Q and an XML database D, assume that $<A_i, A_j, B_{i'}, B_{j'}>$ is a matching cross for D with respect to Q. If the number of distinct elements A_k, where $i \leq k \leq j$ and $FM(A_k,B)=B_{k'}$ $i' < k'$, is no more than the height of D, then we say that A has a bounded matching cross (BMC) with B, otherwise it is unbounded matching cross (UMC).* (See Fig 7)□

Since the number of distinct elements that have the first match after $B_{i'}$ is no more than $|\text{HEIGHT}(D)|$, we can buffer all such A_k's in the main memory and read A_j to find the matching element for $B_{i'}$.

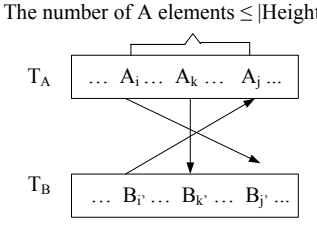


Fig. 7. Illustration to bounded matching cross. The number of elements in T_A between A_i and A_j whose first match is after $B_{i'}$ is no more than the height of the tree.

Example 2: Consider the query and document in Fig 8. $<C_1, C_n, B_1, B_{m'+n-1}>$ is a BMC, because the number of distinct elements C_k ($1 \leq k \leq n$) that has the first match behind B_1 is no more than n, which is bounded by the height of the document, i.e. C has a bounded matching cross with B. In contrast, $<A_1, A_{m+n-1}, B_1, B_{m'+n-1}>$ is a UMC. This is because m or m' is not bounded by the height of the document and thus the number of distinct elements A_k ($1 \leq k \leq m+n$-1) (or similarly B_k, $1 \leq k \leq m'$+n-1) that has the first match behind B_1 (or A_1) is possibly much greater than the height of documents. □

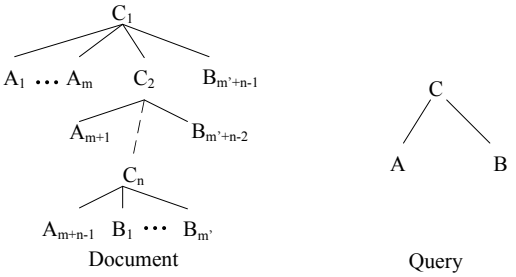


Fig. 8. Bounded matching cross(BMC) and unbounded matching Cross(UMC). $<C_1, C_n, B_1, B_{m'+n-1}>$ is a BMC, while $<A_1, A_{m+n-1}, B_1, B_{m'+n-1}>$ is a UMC.

As shown in Definition 3.3 and Example 2, *matching cross* can be separated to two categories according whether it can be solved by buffering limited elements. In particular, BMC can be solved by buffering bounded number of elements in the main memory. On the other hand, we cannot guarantee to optimally process UMC with limited size of main memory, since it needs us to buffer unbounded number of elements (we say it is unbounded in terms of the height of the document tree).

The following lemma identifies a query class, with respect to which no UMC occurs on any given XML document. In other words, this query class is guaranteed to be processed *optimally* by holistic algorithms. This lemma coincides with the optimal query class in TwigStackList [14].

Lemma 2: Suppose Q is a tree pattern query with only ancestor-descendant (A-D) relationships to connect branching nodes and their children nodes, given any document D, there is no unbounded matching cross (UMC) on D with respect to Q.

PROOF: Details of proof are given in technical report [15].

A natural question is whether all UMC definitely causes the *sub-optimality* of holistic processing algorithms. The answer is "no". Note that query answers of an XML tree pattern usually include only part of query nodes; we can use this observation to identify a larger optimal query class . In order to understand this, let us first consider an XML tree in Figure 9 and an XPath query "$H[.//B]/A$" (A is the selected *return* query node). $<B_1, B_{j+1}, A_1, A_{i+1}>$ is a UMC, since i and j may be greater than the height of XML tree. But we observe that this UMC still can be efficiently processed by registering the information that H_1 has appropriate children B (e.g. B_1) and then scanning B_{j+1} and H_2. Then we can get an exact match (H_2, B_{j+1}, A_1) without outputting any possibly useless intermediate path. This example shows that the existence of

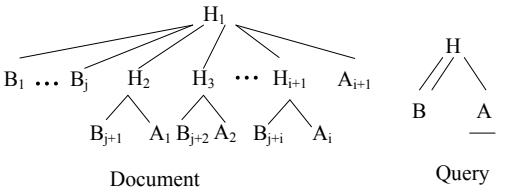


Fig. 9. An example to illustrate unbounded matching cross

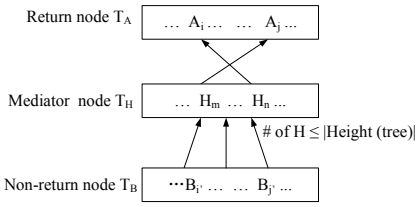


Fig. 10. Illustration to mediator node in UMC. $<B_{i'},B_{j'},A_i,A_j>$ is a UMC. H is mediator node, as the first matches of all elements between $B_{i'}$ and $B_{j'}$ are between H_m and H_n; and the first matches of that between H_m and H_n are A_i and A_j .

UMC does not necessarily result in the *sub-optimality* of algorithm. Some UMC still can be solved by buffering limited information in the main memory. The following definition and lemma show that if there is a *mediator* node in UMC, then such UMC can be still processed optimally.

Definition 3.4: (**mediator in UMC**) *Given a query Q and an XML database D, assume that $<A_i, A_j, B_{i'}, B_{j'}>$ is an unbounded matching cross (UMC) on D with respect to Q, and A is a return node in Q but B is a non-return node. We call the node $H \in Q$ as a mediator node (H may be a return node or not) if the first matches of all elements between $B_{i'}$ and $B_{j'}$ against node H are in the range from H_m to H_n, and the first matches of all elements between H_m and H_n are between A_i and A_j (see Figure 10); and the number of elements between H_m and H_n that are the first matches of B_k ($i' \leq k \leq j'$) is no more than the height of D.* □

For example, consider Fig 9 and the query "$H[.//B]/A$" again. $<B_1,B_{j+1},A_1,A_{i+1}>$ is a UMC, and H is a mediator in this UMC, as the first matches of all elements between B_1 and B_{j+1} against node H is H_1; and the first match of H_1 and H_2 are A_{i+1} and A_1; and $2 \leq$ Height(D).

Because of the existence of *mediator node* in UMC, we still can guarantee the optimality of algorithm by buffering limited elements of *mediator nodes* in the main memory. In the example of Figure 9, we only need to buffer H_1 and H_2 to the main memory and record that H_1 and H_2 have the matching element with node B. Note that we do not need to buffer $B_1, \ldots, B_j$ in the main memory as they are not return nodes.

The next definition and lemma identify a subclass of tree pattern queries, with respect to which, given any XML document, we can always find a mediator node in a UMC.

Definition 3.5: (**mediator subclass**) *We say that a query Q belongs to mediator subclass if and only if given any return node N in Q and a branching node B in the path from N to the root, there are only ancestor-descendant relationships between B and its children that are not in the path from N to the root.*

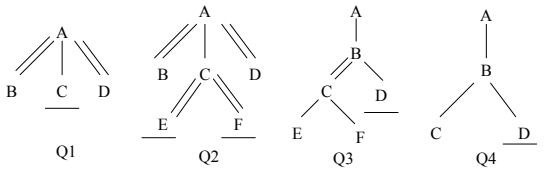


Fig. 11. Example queries to illustrate mediator subclass. Q1,Q2,Q3 is in mediator subclass, but Q4 not, because of (B,C) edge.

For example, Figure 11 shows four example queries. Q1, Q2 and Q3 belong to mediator subclass, but Q4 does not becuase of (B, C) edge.

Lemma 3: Given a query Q that is in mediator subclass and a document D, for each UMC in D against Q, there exists a mediator node $H \in Q$ in this UMC.

PROOF: Details of proof are given in technical report [15].

In the next section, we will develop a holistic algorithm to process mediator subclass query optimally.

As a final remark of this section, it is important to note that the properties shown in the above theorems is independent of (i) any concrete labeling schemes and (ii) any special data index structures, such as XB tree[3], XR tree [11] and R tree [7]. This is because (i) the proof of the above theorems does not rely on any specific labeling scheme, and (ii) while special index structure can skip elements to accelerate processing in holistic XML query processing, these index structures *cannot* achieve the larger optimal query class, as the main bottleneck of optimality is the size of main memory.

4 HOLISTIC ALGORITHMS

In this section, we propose an algorithm to evaluate an extended XML tree query. The challenge in the algorithm is to achieve a large optimal query class according to aforementioned theorems.

4.1 TreeMatch for $Q^{/,//,*}$

4.1.1 Data structures and notations

There is an input list T_q associated with each query node q, in which all the elements have the same tag name q. Thus, we use e_q to refer to these elements. $cur(T_q)$ denotes the current element pointed by the cursor of T_q. The cursor can be advanced to the next element in T_q with the procedure $advance(T_q)$.

There is a set S_q associated with each *branching* query node q (not each query node). Each element e_q in sets consists of a 3-tuple $(label, bitVector, outputList)$. $label$ is the *extended Dewey* label of e_q. $bitVector$ is used to demonstrate whether the current element has the proper children or descendant elements in the document. Specifically, the length of $bitVector(e_q)$ equals to the number of child nodes of q. Given a node $q_c \in children(q)$, we use $bitVector(e_q)[q_c]$ to

denote the bit for q_c. Specifically, $bitVector(e_q)[q_c]$="1" if and only if there is an element e_{q_c} in the document such that the e_q and e_{q_c} satisfy the query relationship between q and q_c. Finally, $outputList$ contains elements that potentially contribute to final query answers. Next, we introduce two properties of elements in *outputList* and *bitVector* in details.

At every point during the computing, for each element e_q in set S_q, (i) if all bits in $bitVector(e_q)$ are "1", then e_q is guaranteed to match the subtree rooted with q. Therefore, if q is the root, then e_q is guaranteed to match the whole query, and (ii) element $e \in outputList(e_q)$ is the query answer if and only if e_q matches the whole tree query. Therefore, using both properties, we say that whether an element $e \in outputList(e_q)$ is a query answer can be *accurately* reflected by the corresponding $bitVector(e_q)$, illustrated as follows.

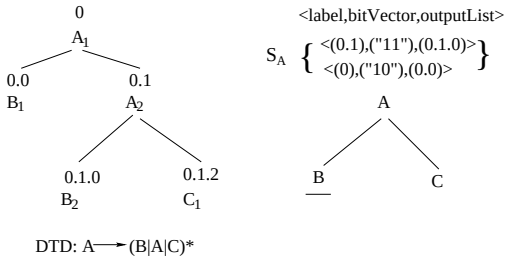


Fig. 12. Illustration to set encoding (the left side is an XML tree and the right one is a query with running-time set encoding)

Example 3: Figure 12 illustrates the set encoding S_A to query node A for an example document. There are two tuples in set S_A. Since A_1 ("0") has only one child B_1 and no child element to match C, $bitVector(A_1)$="10". In contrast, $bitVector(A_2)$="11", since A_2 ("0.1") has two children B_2 and C_1, which satisfy the P-C relationships in the query. Since all bits in $bitVector(A_2)$ are "1", B_2("0.1.0") is guaranteed to be a query answer. □

In our algorithm, we will frequently use the following two notations. (1) $\texttt{NAB}(q)$ denotes the Nearest Ancestor Branching node of q in the query pattern Q. Formally, q'=$\texttt{NAB}(q)$ if and only if q' is a branching node and q' is an ancestor of q and there is no other branching node q'' s.t. q'' is in the path from q' to q. If there is no such ancestor of q, then $\texttt{NAB}(q)$ denotes the top branching node in query. (2) $\texttt{NDB}(q)$ denotes the nearest descendants branching (or leaf) nodes of q. Formally, $q' \in \texttt{NDB}(q)$ if and only if q' is a branching or leaf node and q' is a descendant of q and there is no other branching or leaf node q'' s.t. q'' is in the path from q' to q. For example, see the query Q3 in Figure 11, $\texttt{NAB}(E)=\{C\}$, $\texttt{NAB}(D)=\{B\}$, $\texttt{NDB}(B)=\{C, D\}$.

4.1.2 Intuitive example

Before we formally introduce the algorithm TreeMatch, let us first see an example to intuitively understand this algorithm. Here the key point is set encoding of elements.

Example 4: Consider the data and query in Figure 12 again. Note that B is the single return node. Firstly, B_1 and C_1 are read. Since A_1 now has only one child B_1 and one descendant C_1 (not child), we insert A_1 to set S_A and *bitVector*(A_1)="10" (see Table 1). Next, when B_2 and C_1 are read, since A_2 has two children B_2 and C_1, we add A_2 to set and *bitVector*(A_2)="11". Finally, we empty set S_A and output one element B_2 in the *outputlists*. Note that unlike previous algorithms such as TwigStack [3] and TJFast [16], *bitVector* is used to accurately record matching results, thus leading to avoiding the output of B_1, as *bitVector*(A_1) is "10". But TwigStack and TJFast would output two "useless" elements A_1 and B_1 in that case, and therefore, entail more I/O cost. □

Current elements	Set encoding of S_A
B_1,C_1	<0,"10",0.0>
B_2,C_1	<0.1,"11",0.1.0>, <0,"10",0.0>

TABLE 1
Set encoding for the example in Figure 12

Algorithm 1: Algorithm TreeMatch for class $Q^{/,//,*}$

```
1: locateMatchLabel(Q);
2: while (¬end(root)) do
3:     f_act = getNext(topBranchingNode);
4:     if (f_act is a return node)
5:         addToOutputList(NAB(f_act), cur(T_{f_act}));
6:     advance(T_{f_act}); // read the next element in T_{f_act}
7:     updateSet(f_act); // update set-encoding
8:     locateMatchLabel(Q); // locate next element with
       matching path
9: emptyAllSets(root);
```

4.1.3 TreeMatch

Now we go through Algorithm 1. Line 1 locates the first elements whose pathes match the individual root-leaf path pattern. In each iteration, a leaf node f_{act} is selected by $getNext$ function (line 3). The purpose of line 4, 5 is to insert the potential matching elements to $outputlist$. Line 6 advances the list $T_{f_{act}}$ and line 7 updates the set encoding. Line 8 locates the next matching element to the individual path. Finally, when all data have been processed, we need to empty all sets in Procedure *EmptyAllSets* (Line 9) to guarantee the completeness of output solutions.

In Procedure $addToOutputList(q, e_{q_i})$, we add the potential query answer e_{q_i} to the set of S_{e_q}, where q is the nearest ancestor branching node of q_i (i.e. $NAB(q_i) = q$). Procedure $updateSet$ accomplishes three tasks. First, clean the sets according to the current scanned elements. Second, add e into set and calculate the proper *bitVector*. Finally, we need recursively update the ancestor set of e. Because of the insertion of e, the *bitVector* values of ancestors of q need update.

Algorithm *getNext*(see Algorithm 2) is the core function called in TreeMatch, in which we accomplish two tasks. For the first task to identify the next processed node, Algorithm *getNext(n)* returns a query leaf node f according to the following recursive criteria (i) if n is a leaf node, f=n(line 2); else (ii) n is a branching node, then suppose element e_i matches node

Algorithm 2: Procedures and Functions in TreeMatch

1 Procedure locateMatchLabel(Q)
1: for each leaf $q \in Q$, locate the extended Dewey label e_q in list T_q such that e_q matches the individual root-leaf path

Procedure addToOutputList(q, e_{q_i})
1: **for** *each* $e_q \in S_q$ **do**
2: **if** (satisfyTreePattern(e_{q_i},e_q))
3: $outputList(e_q).add(e_{q_i})$;

Function satisfyTreePattern(e_{q_i},e_q)
1: **if** ($bitVector(e_q, q_i)$ = '1') return true;
2: **else** return false;

Procedure updateSet(q, e)
1: $cleanSet(q, e)$;
2: add e to set S_q; //set the proper $bitVector(e)$
3: **if** ($\neg isRoot(q) \wedge (bitVector(e)$="1...1"))
$updateAncestorSet(q)$;

Procedure cleanSet(q, e)
1: **for** each element $e_q \in S_q$ **do**
2: **if** (satisfyTreePattern(e_q,e))
3: **if** (q is a return node)
4: $addToOutputList(NAB(q),\ e)$;
5: **if** ($isTopBranching(q)$)
6: **if** (there is only one element in S_q)
7: output all elements in $outputList(e_q)$;
8: **else** merge all elements in $outputList(e_q)$ to $outputList(e_a)$, where e_a=NAB(e_q);
9: delete e_q from set S_q;

Procedure updateAncestorSet(q)
1: /*assume that q' = NAB(q)*/
2: **for** each $e \in S_{q'}$ **do**
3: **if** ($bitVector(e,\ q) = 0$)
4: $bitVector(e, q) = 1$;
5: **if** ($\neg isRoot(q) \wedge (bitVector(e)$="1...1"))
6: $updateAncestorSet(q')$;

Procedure emptyAllSets(q)
1: **if** (q is not a leaf node)
2: **for** each child c of q **do** EmptyAllSets(c);
3: **for** each element $e \in S_q$ **do** cleanSet(q, e);

Algorithm 3: getNext(n)

1: **if** (isLeaf(n)) **then**
2: return n
3: **else**
4: **for each** $n_i \in$ NDB(n) **do**
5: $f_i = getNext(n_i)$
6: **if** (isBranching(n_i) $\wedge$ $\neg empty(S_{n_i})$)
7: return f_i
8: **else** $e_i = max\{p | p \in MB(n_i, n)\}$
9: **end for**
10: max = $maxarg_i\{e_i\}$
11: **for each** $n_i \in$NDB(n) **do**
12: **if** ($\forall e \in MB(n_i, n) : e \notin ancestors(e_{max})$)
13: return f_i;
14: **endif**
15: **end for**
16: min = $minarg_i\{f_i | f_i\ is\ not\ a\ return\ node\}$
17: **for each** $e \in$ MB(n_{min}, n)
18: **if** ($e \in ancestors(e_{max})$) updateSet(S_n, e)
19: **end for**
20: return f_{min}
21: **end if**

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

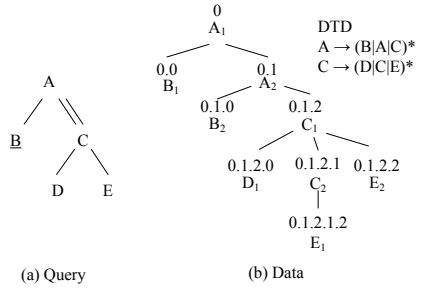


Fig. 13. Illustration to Algorithm TreeMatch for class $Q^{/,//,*}$

n in the corresponding path solution(if more than one element that matches n, e_i is the *deepest* one by level)(line 7,8), we return f_{min} such that the current element e_{min} in $T_{f_{min}}$ has the minimal label in all e_i by lexicographical order(line 13,20).

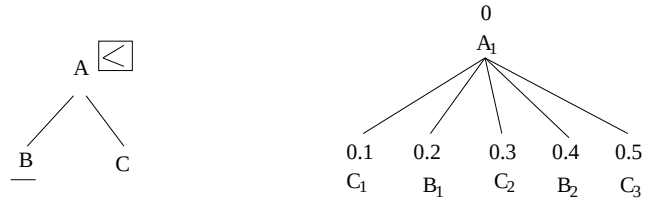
For the second task of *getNext*, before an element e_b is inserted to the set S_b, we ensure that e_b is an ancestor (or parent) of each other element e_{b_i} to match node b in the corresponding path solutions(line 13). If there are more than one element to match the branching node b, e_{b_i} is defined as their *deepest*(i.e. maximal) element(line 8).

Example 5: We use the query and document in Figure 13 to illustrate TreeMatch algorithm. Table 2 demonstrates the current access elements, the sets encoding and the corresponding output elements. There are two branching nodes in the query. Firstly, B_1, D_1 and E_1 are scanned. C_1 and C_2 are added into the set S_C, but their bitVectors is "10" and "01", which indicate that C_1 and C_2 have only one child respectively. In this scenario, recall that TJFast may output path solutions $A_1/A_2/C_1/D_1$ and $A_1/A_2/C_1/C_2/E_1$, which might be useless to final results. Thus, our algorithm TreeMatch diminishes the unnecessary I/O cost. Next, E_2 is scanned and the bitVector(C_1) becomes "11" as C_1 has two children now. Similarly, the bitVector(A_1) is "11" too. In this moment, we guarantee that A_1 matches the whole pattern tree, as all bits in bitVector(A_1) is 1 (Lemma 4.1 generalizes this observation.) Finally, when B_2 is scanned, A_2 is added to set S_A. Therefore, Treematch outputs two final results B_1 and B_2. Note that there are no useless nodes output here. □

Through this example, we illustrates two differences between TJFast and TreeMatch. (1) TJFast outputs one useless

Current elements	Set encoding S_A	Set encoding S_C
B_1,D_1,E_1	<0,"10",Ø>	<0.1.2,"10",Ø>, <0.1.2.1,"01",Ø>
B_1,D_1,E_2	<0,"11","0.0">	<0.1.2,"11",Ø>, <0.1.2.1,"01",Ø>
B_2,D_1,E_2	<0,"11","0.0"> <0.1,"11","0.1.0">	<0.1,"11",Ø>, <0.1.2.1,"01",Ø>

TABLE 2
Set encoding for the example in Figure 13



(a) Ordered based query (b) Document

Fig. 14. An example ordered XML tree pattern query. When we scan B_1 and C_1, we do not insert C_1 to *minChild*, as it is before B_1 and does not satisfy the order condition of query.

Current elements	Set encoding S_A	
	minChild	maxChild
B_1,C_1	(0.2,Ø)	(0.2,Ø)
B_1,C_2	(0.2,0.3)	(0.2,0.3)
B_2,C_2	(0.2,0.3)	(0.4,0.3)
B_2,C_3	(0.2,0.3)	(0.4,0.5)

TABLE 3
Partial set encoding for the example in Figure 14

intermediate path $A_1/A_2/C_1/C_2/E_1$, but TreeMatch uses the bitVector encoding to solve this problem. (2) TJFast outputs the path solution for all nodes in query, but TreeMatch only outputs nodes for return nodes (i.e. node B in the query) to reduce I/O cost.

When there are multiple return nodes in a query, TreeMatch produces the corresponding *outputList* for each of them, and then outputs the individual solution for each return node, and merges all these solutions to get the final result bindings. It is important to note the differences between TreeMatch and TJFast [16]. Even if the available amount of main memory is large, TJFast possibly outputs many path solutions that do not contribute to any final answers. However, TreeMatch can efficiently use these available main memory (by buffering potential useful elements in *outputlist*) to guarantee that each output element contributes to final answers. Therefore, TreeMatch not only identifies a larger optimal query class than TJFast, but also has the ability to fully utilize the available amount of the main memory (which will be verified in our experiments).

4.2 Extension for order-based queries $Q^{/,//,*,<}$

In this section we extend TreeMatch algorithm to support ordered-based queries $Q^{/,//,*,<}$. In order to record the position information of elements, we add *minChild* and *maxChild* attributes for each tuple in sets. That is, each tuple in sets now is a 5-tuple: $<label, bitVector, outputList, minChild, maxChild>$. The length of *minChild*(e_q) and *maxChild*(e_q) is equal to the number of children of q. Assume that q_1, ..., q_n are the children node of q (in order) in the query. Given an element $e_{q_i}^{min}$ in *minChild*(e_q) and $e_{q_i}^{max}$ in *maxChild*(e_q), $e_{q_i}^{min}$ is the minimal element that is greater than the element $e_{q_{i-1}}^{min}$ (if any) and $e_{q_i}^{max}$ is the maximal element that is smaller than $e_{q_{i+1}}^{max}$ (if any). In particular, $e_{q_1}^{min}$ is the left-most children of e_q, and $e_{q_n}^{max}$ is the right-most children.

Example 6: See the query and document in Figure 14. Table 3 shows the values of *minChild* and *maxChild* attributes in set. (Note that the full presentation of each element in S_A is a 5-tuple. Here we only show *minChild* and *maxChild* for the purpose of this example.) When B_1 and C_1 are read, since C_1 is before B_1, we do not insert C_1 as a *minChild*, as it is not greater than B_1. Only after C_2 is read, we insert C_2 to *minChild*. When B_2 and C_3 are scanned, they become the respective *maxChild* for node B and C. □

Algorithm 4 describes the extended TreeMatch algorithm for answering ordered tree queries. The purpose of the extension is to maintain and check the order relationship among the matching elements of query sibling nodes. In line 2 of Procedure updateSet, we need to set the proper *minChild* and *maxChild* according to the current elements. In Function satisfyTreePattern, we also need to check the order restriction according to *minChild* and *maxChild*.

Although the frequent updates of *minChild* and *MaxChild* values in sets may incur CPU cost, compared to the reduction of useless intermediate results, as we will see in the experimental evaluation, those extra CPU cost is worthwhile.

4.3 Extension for queries with negative edges $Q^{/,//,*,<,\neg}$

In this section we further extend TreeMatch to support negative edges (see algorithm 5). We add *negBitVector* to record the matching information about negative child edge. Given a node $q_c \in negativeChildren(q)$, $negBitVector(e_q)[q_c]$="0" if and only if there is no element e_{q_c} in the document such that the e_q and e_{q_c} satisfy the query relationship in between q and q_c. In this way, in order to know whether all negative children of q are satisfied, we only check whether all children's *negBitVectors* are "0". In line 2 of Procedure updateSet, we need to set the proper *negBitVector* according to the current elements. In Function satisfyTreePattern, e_q is a valid element only if the *negBitVector* is '0'.

4.4 Analysis of algorithms

In this section, we discuss the correctness of TreeMatch, and then analyze its complexity.

Lemma 4: In Algorithm TreeMatch, suppose any element e_q is popped from set S_q, where q is the top branching node in Procedure CleanSet(q), then e_q matches the whole query if and only if all bits in $bitVector(e_q)$ are "1" and all

Algorithm 4: Algorithm TreeMatch for class $Q^{/,//,*,<}$

```
1 Procedure updateSet(q, e)
    ...
    2: add e to set S_q; //set the proper bitVector, minChild and
    maxChild
    ...
  Function satisfyTreePattern(q_i,e_q)
    1: assume that child nodes of q in Q are q_1, ..., q_n (in order)
    2: if (e_{q_i} < minChild(e_q, q_{i-1})) return false;
    3:     else if (e_{q_i} > maxChild(e_q, q_{i+1})) return false;
    4:         else if (bitVector(e_q, q_i) = '1')     return true;
    5:             else return false;
```

Algorithm 5: Algorithm TreeMatch for class $Q^{/,//,*,<,\neg}$

```
1 Procedure updateSet(q, e)
    ...
    2: add e to set S_q; //set the proper bitVector, negBitVector,
    minChild and maxChild
    ...
  Function satisfyTreePattern(q_i, e_q)
    1: if (e_{q_i} < minChild(e_q, q_{i-1})) return false;
    2:     else if (e_{q_i} > maxChild(e_q, q_{i+1})) return false;
    3:         else if ((bitVector(e_q)[q_i]) = '1') and
    (negBitVector(e_q)[q_i]) = '0'))
    4:             return true;
    5:         else return false;
```

children of e_q satisfy order condition (if any), and all bits in $negBitVector(e_q)$ are "0" (if any). □

Lemma 5: In Algorithm TreeMatch, suppose any element e_q is popped from set S_q, where q is the top branching node in Procedure CleanSet(q), then e_q matches the whole query if and only if all elements in outputList(e_q) belong to final query answers. □

Using Lemma 4 and 5, we can see that whether or not an element is a query answers is exactly reflected by the values of the corresponding *bitVector*, $negBitVector$ and *minChild*, *maxChild*. Further, by Line 5-7 in Procedure *CleanSet*, all correct solutions are output. In addition, each matching element is guaranteed to be inserted to the related sets in Procedure *addToOutputList*. Thus, the output solutions are also *complete*. Therefore, we have the following result.

Theorem 6: Given an extended tree pattern query Q and an XML database D, Algorithm TreeMatch correctly returns all the answers for Q on D. □

While the correctness holds for any given query, the I/O optimality holds only for a subset of extended query class. In these cases, TreeMatch guarantees that each output element in Procedure CleanSet belongs to final query solutions. Next, we show the corresponding optimality query subclass for three categories of queries, i.e. $Q^{/,//,*}$, $Q^{/,//,*,<}$ and $Q^{/,//,*,<,\neg}$.

Theorem 7: Consider an XML database D and an extended tree pattern query $Q^{/,//,*}$ in mediator subclass *(defined in Definition 3.5), the worst case I/O complexity of* TreeMatch *is linear to the sum of the sizes of input and results. The worst-case memory space complexity is $O(d^2 * b + d * f)$, where f is the number of leaves in Q, d is the length of the longest label in the input lists and b is the number of branching nodes.*

PROOF (Sketch): Given any return node q in Q, let $b = NBA(q)$, according to Definition 3.5, all edges except (b, q) between b and its children are ancestor-descendant relationships. b is a mediator node for any UMC involving in q. In Procedure addToOutputList, when each element e_q is inserted to outputList, it is guaranteed to satisfy the subtree rooted with q (Line 2). In Procedure UpdateAncestorSet, the elements in outputList is moved to its ancestor set only if the current subtree is satisfied. We recursively guarantee that each e_q in outputList satisfies the whole tree pattern. Therefore, each element e_q is inserted to outputList of e_b only if e_b satisfies the whole tree pattern. We can safely write each element in outputList to disk in Procedure CleanSet and thus the worst case I/O complexity of TreeMatch is linear to the sum of the sizes of input and results. Finally, as for space complexity, the number of elements in each set S is at most d, where d is the length of the longest label in the input lists and thus the total space complexity of d labels is $O(d^2)$. Note that each element in outputList guarantees to contribute to the final results, and it may be written to the secondary storage and thus their size is not calculated here. □

For queries with ordered node (i.e. $Q^{/,//,*,<}$), we can identify a larger optimal class. If node q is an order node in Q, the parent-child relationship between q and its first child does not affect the optimality of TreeMatch. Intuitively, this is because the order restriction stops some unbounded matching cross from happening.

Definition 4.1: (optimal subclass for $Q^{/,//,*,<}$) *We say that a query Q belongs to the optimal subclass for $Q^{/,//,*,<}$ if and only if the parent-child relationship of Q occurs only in the following edges E, (1) given any return node q in Q, E is in the path from q to $root$; or (2) let $E = (a, b)$, then a should be an ordered node and b is the first child of a.*

Theorem 8: Consider an XML database D and an extended tree pattern query $Q^{/,//,*,<}$ in the subclass defined in Definition 4.1, the worst case I/O complexity of TreeMatch *is linear to the sum of the sizes of input and results. The worst-case memory space complexity is $O(2d^2 * b + d * f)$, where f is the number of leaves in Q, d is the length of the longest label in the input lists and b is the number of branching nodes.*

PROOF (Sketch): We need to show that the parent-child (P-C) relationship in the first branching edge of an ordered node does not affect the optimality of TreeMatch. Details of proof are given in technical report [15].

For queries with ordered nodes and negative edges (i.e. $Q^{/,//,*,<,\neg}$), the following results show that the existence of parent-child (or ancestor-descendant) edges in any negative edges does not affect the optimality of TreeMatch. Intuitively, this is because parent-child relationships in negative edges do not cause the matching cross.

Definition 4.2: (optimal subclass for $Q^{/,//,*,<,\neg}$) *We say that a query Q belongs to the optimal subclass for $Q^{/,//,*,<,\neg}$ if and only if the parent-child relationship of Q occurs only in the following edges E, (1) given any return node q in Q,*

E is in the path from q to root; or (2) let $E = (a, b)$, then all child nodes of a are ordered and b is the first child of a; or (3) E is a negative edge.

Theorem 9: Consider an XML database D and an extended optimal tree query $Q^{/,//,*,<,\neg}$ defined in Definition 4.2, the worst case I/O complexity of TreeMatch *is linear to the sum of the sizes of input and results. The worst-case memory space complexity is $O(2d^2 * b + d * f)$, where f is the number of leaves in Q, d is the length of the longest label in the input lists and b is the number of branching nodes.*

PROOF (Sketch): We need to show the existence of negative P-C and A-D relationship does not affect the optimality of TreeMatch. Details of proof are given in technical report [15].

5 EXPERIMENTS

In this section, we present an extensive experimental study of TreeMatch on real-life and synthetic data sets. Our results verify the effectiveness, in terms of accuracy and optimality, of the TreeMatch as holistic twig join algorithms for large XML data sets. These benefits become apparent in a comparison to previously four proposed algorithms TwigStack [3], TJFast [16], OrderedTJ [17] and TwigStackListNot [26]. The reason that we choose these algorithms for comparison is that (1) similar to TreeMatch, both TJFast and TwigStack are two holistic twig pattern matching algorithms. But they cannot process queries with order restriction or negative edges; and (2) OrderedTJ is a holistic twig algorithm which can handle order-based XML tree pattern, but is not appropriate for queries with negative edges; and finally (3) TwigStacklistNot is proposed for queries with negative edges, but it can not work for ordered queries. Only TreeMatch algorithm can process queries with order restriction, negative edge and wildcards.

5.1 Experiment Settings and Dataset

We implemented all tested algorithms in JDK 1.4 using the file system as a simple storage engine. We conducted all the experiments on a computer with Intel Pentium IV 1.7GHz CPU and 2G of RAM. To offer a comprehensive evaluation of our new algorithms, we conducted experiments on both synthetic and real XML data. The synthetic dataset is generated randomly. There are totally 7 tags A,B,...,F,G in the dataset and tags are assigned uniformly from them. The real data are DBLP (highly regular) and Treebank (highly irregular), which are included to test the two extremes of the spectrum in terms of the structural complexity. The recursive structure in TreeBank is deep (average depth: 7.8, maximal depth: 36). We can easily find queries on this dataset to demonstrate the sub-optimality for our tested algorithms.

5.2 Query class $Q^{/,//,*}$

In this section, we show the experimental results for queries class $Q^{/,//,*}$. All queries tested in our evaluation are shown in Figure 15 and 16.

Small size of main memory In the first experiment, we did not allow the *outputlist* in TreeMatch to buffer any elements in the main memory, meaning that any element added to *outputlist* should be output to the secondary storage. Then the requirement for main memory size is quite small. The purpose of this experiment is to simulate the application where the document is extremely large but the available main memory is relatively small. Table 4 shows the number of total output elements (including intermediate and final results) and the corresponding percentage of useful elements. We made the experiments by using three different sizes of random documents. In particular, D1 has 100K nodes and D2 has 500K nodes and D3 has 1M nodes. From Table 4, we observe that for most of queries, TreeMatch achieves the optimality in the sense that each of the output elements does belong to final results. The only exception is in Q3 and Q6, where according to Theorem 7, we cannot guarantee the optimality. Interestingly, Q6 is optimal for D1 and D2, but only slightly sub-optimal for D3. This can be explained that D3 is a larger document than D1 and D2 so that D3 manifests the sub-optimality which is hidden in D1 and D2. Figure 17(a) compares the performance of TreeMatch with other three existing algorithms. Clearly, TreeMatch is the best for all queries. This advantage is due to the fact that TreeMatch guarantees that (almost) all of output elements belong to final results, which, in general, avoids the I/O cost for outputting useless intermediate results.

Large size of main memory In the second experiment, we allow the *outputlist* to buffer all elements in the main memory. The purpose of this experiment is to simulate the application where the available main memory is large so that

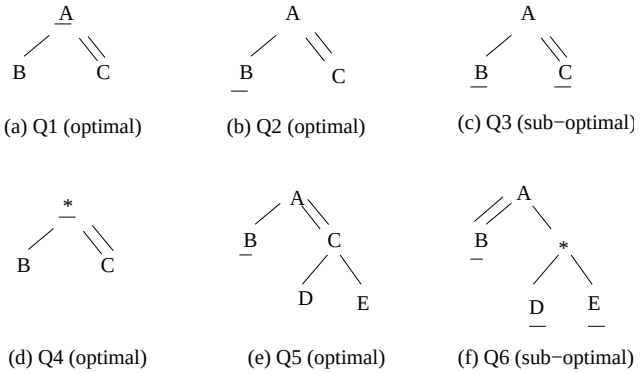


Fig. 15. Queries for random data

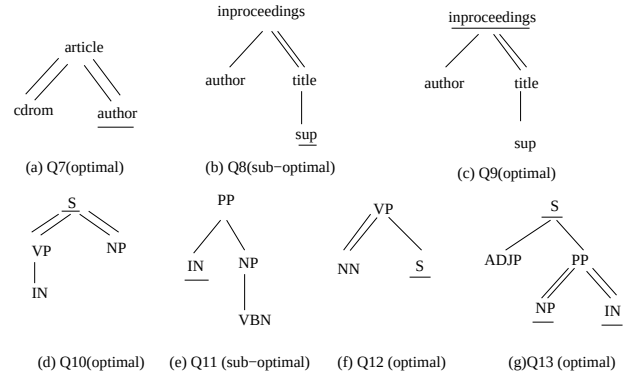
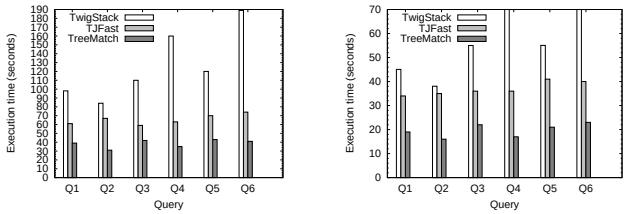
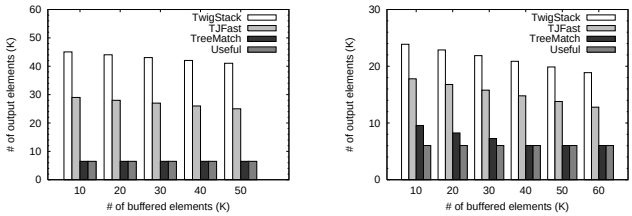


Fig. 16. Queries for DBLP (Q7-Q9) and TreeBank (Q10-Q13) data



(a) Small memory (b) Large memory

Fig. 17. Execution time of $Q^{/,//,*}$ on random data



(a) Q1 (b) Q6

Fig. 19. Output data size with varying memory (medium memory)

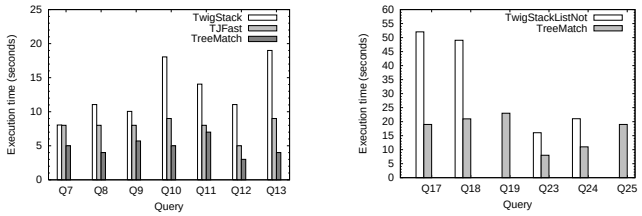
a big portion of documents can fit in the main memory. Table 5 shows the maximal number of elements buffered in order to avoid outputting any useless intermediate results. An obvious observation is that Q3 and Q6 need to buffer many elements, but all other queries only need to buffer very small number of elements. This also can be explained that all queries except Q3,Q6 belong to the optimal query class. We compared the performance of three algorithms in Figure 17(b) and Figure 18(a). Obviously, TreeMatch is superior to TwigStack and TJFast, reaching $20\%-95\%$ improvement in execution time for all queries.

Query	D1		D2		D3	
	O	P	O	P	O	P
Q1	1321	100%	6576	100%	13290	100%
Q2	3558	100%	17757	100%	35649	100%
Q3	9575	98.8%	95291	99.9%	156954	94.5%
Q4	6635	100%	33055	100%	65691	100%
Q5	296	100%	1313	100%	2782	100%
Q6	7506	100%	94132	100%	127478	99.9%

TABLE 4
Number of output elements (O) and the percentage (P) of useful elements for TreeMatch on random data

	D_1	D_2	D_3
Q1	5	6	6
Q2	9	10	11
Q3	528	27067	89779
Q4	6	7	8
Q5	7	8	10
Q6	520	26808	89627

TABLE 5
# of required buffered elements (Random data)



(a) $Q^{/,//,*}$ (b) $Q^{/,//,*,<,\neg}$

Fig. 18. Execution time on DBLP, TreeBank data (large memory)

Medium size of main memory In most real application, the main memory size is not so large that the whole document can fit in memory, neither so limited that only the elements in a single path can load in memory. In order to test whether TreeMatch has the ability to fully exploit the available medium size of main memory, we show the performance of algorithms in terms of the number of output elements with varying the size of main memory in Figure 19. In this experiment, we choose $Q1$ and $Q6$, since $Q1$ is an optimal query for TreeMatch, but $Q6$ is sub-optimal. The experimental results show that the number of output elements in TreeMatch is always much less than that in TwigStack and TJFast for all sizes of main memory. In particular, for $Q1$, with the increasing of the size of the available main memory, the number of output elements in TwigStack and TJFast decreases linearly. The reason is that TwigStack and TJFast buffer the intermediate results in the main memory and reduce the output of intermediate results. But the numbers of output elements in TreeMatch remain the same, which always equals the final result size. For query $Q6$, all algorithms are not optimal. But TreeMatch still outputs much less elements than TwigStack and TJFast.

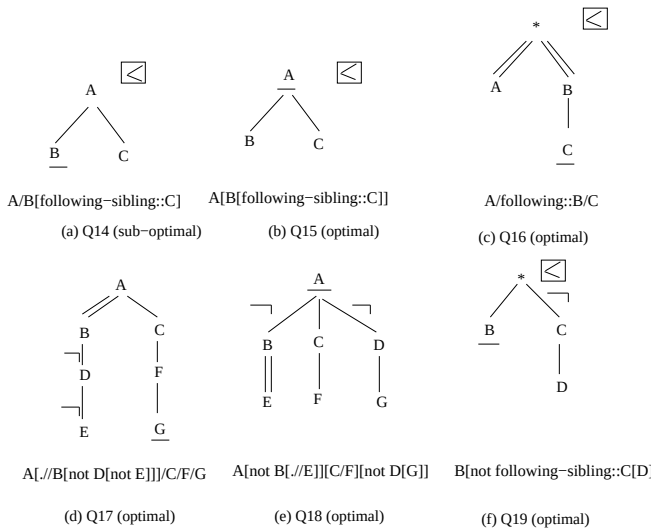
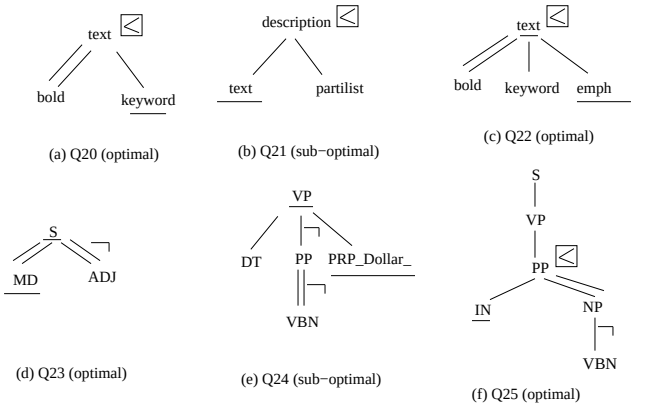


Fig. 20. Queries for class $Q^{/,//,*,<,\neg}$

5.3 Query class $Q^{/,//,*,<,\neg}$

In this section, we show the experimental results for queries class $Q^{/,//,*<,\neg}$, which may contain order restrction, negative edge and wildcards. The tested queries are shown in Figure 20 and 21. Table 6 shows the number of output elements and



Query	XPath expressions
Q20	text//bold/following−sibling::keyword
Q21	//description/partilist/preceding−sibling::text
Q22	//text//bold[following−sibling::keyword[following−sbling::emph]]
Q23	S[not .//ADJ]//MD
Q24	VP[DT][not PP[not .//VBN]]/PRP_DOLLAR_
Q25	S/VP/PP/IN[following−sibling::NP[not VBN]]

Fig. 21. Query for DBLP and TreeBank data

Query	D1		D2		D3	
	O	P	O	P	O	P
Q14	3596	68.2%	17922	69.8%	35959	68.7%
Q15	2481	100%	12367	100%	24575	100%
Q16	1075	100%	5408	100%	10820	100%
Q17	19792	100%	100008	100%	199727	100%
Q18	3926	100%	20182	100%	39796	100%
Q19	19565	100%	190789	100%	246783	100%

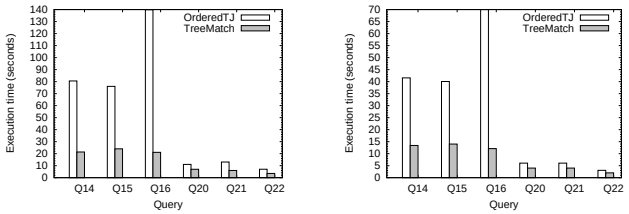
TABLE 6
# of output elements (O) and the percentage (P) of useful elements for TreeMatch on random data

the number of final query answers in the case of small main memory against synthetic data sets. For all optimal queries (i.e. Q15-Q19), the number of output elements is the same as that of final results. This result verifies the correctness of theorems about the optimality of TreeMatch algorithm.

Finally, we made experiments on the DBLP and TreeBank with queries in Figure 21. Since Q17,Q18,Q23,Q24 have negative edges, we compare TreeMatch with TwigStackListNot[26] in Figure 18(b). In addition, as Q14-Q16, Q20-Q22 are order-based queries, we compare TreeMatch with OrderedTJ[17] in Figure 22. From all tested queries, TreeMatch has better performance than the previous algorithms. We contribute this improvement to the larger optimal query class TreeMatch algorithm achieves. Finally, as for queries Q19 and Q25, since two queries contain wildcards, negative edge and order restriction, only our TreeMatch can answer such complicated queries. The execution times of Q19 and Q25 are 16 and 12 seconds, respectively. Note that the above execution performance is achieved by using a relatively very small buffer size, we expect that our system can scale well for even gigabytes of XML data based on the current machine.

	D_1	D_2	D_3
Q14	3926	20182	39796
Q15	9	9	10
Q16	4	5	6
Q17	3	5	6
Q18	6	8	9
Q19	9	11	11

TABLE 7
# of required buffered elements (Random data)



(a) Small memory (b) Large memory

Fig. 22. Execution time of $Q^{/,//,*,<}$ on random data

6 Related Work

In the context of semi-structured and XML databases, tree-based query pattern is a very practical and important class of queries. Lore DBMS [9] and Timber [10] systems have considered various aspects of query processing on such data and queries. XML data and various issues in their storage as well as query processing using relational database systems have recently been considered in [18], [27], [22], [19]. Our holistic algorithm TreeMatch for extended tree patterns can leverage these previous techniques.

From the aspect of theoretical research about the optimality of XML tree pattern matching, Choi et al. [8] developed theorems to prove that it is impossible to devise a holistic algorithm to guarantee the optimality for queries with any combination of P-C and A-D relationships. Shalem et al. [21] researched the space complexity of processing XML twig queries. Their paper showed that the upper bound of full-fledge queries with parent-child and ancestor-descendant edges are $O(D)$, where D is the document size. In other words, their results also theoretically prove that there exists no algorithm to optimally process an arbitrary query $Q^{/,//,*}$. Our research in this article moves the frontier forward by identifying a large subclass of $Q^{/,//,*}$, which can be guaranteed to process optimally.

The recent papers (e.g. [17], [26], [6], [5]) are also closely related to ours. In paper [17], a new holistic algorithm, called OrderedTJ, is proposed to process order-based XML tree query. In paper [26], an algorithm called TwigStackListNot is proposed to handle queries with negation function. Note that the optimal query classes identified in those papers are smaller than that in this article. Chen et al [6] proposed different data streaming schemes to boost the holism of XML tree pattern processing. They showed that larger optimal class can be achieved by refined data streaming schemes. We believe that our work is orthogonal and complementary to their work. This is because based on the theorems on "matching

cross" in this paper, their algorithm iTwigJoin [6] can be further enhanced to identify a larger optimal query class with different streaming schemes. In addition, Twig^{2}Stack [5] is proposed for answering generalized XML tree pattern queries. Note the difference between *generalized* XML tree pattern and *extended* XML tree pattern here. Generalized XML tree pattern is defined to include optional axis which models the expression in `LET` and `RETURN` clauses of XQuery statements. But extended XML tree pattern is defined to include some complicated conditions like negative function, wildcard and order restriction.

Besides the holistic algorithms, there are other approaches to match an XML tree pattern, such as ViST ([24], [23]) and PRIX ([20]), which transform an XML tree pattern match to sequence match. Their algorithms mainly focus on ordered queries, and it is non-trivial to extend those methods to handle unordered queries and extended queries studied in this article. Note that the paper [18] made comprehensive experiments to compare different XML tree query processing algorithms (including sequence match and holistic match) and concluded that the family of holistic processing methods, which provides performance guarantees, is the most robust approach. In this article, we follow the line of holistic XML tree pattern processing and give a complete solution to efficiently process extended XML tree queries with wildcards, negative predicates and ordered/unordered restriction.

7 Conclusions

We have introduced a notion of *matching cross* to address the problem of the sub-optimality in holistic XML tree patten matching algorithms. We have identified a large optimal query classes for three kinds of queries, that is $Q^{/,//,*}$, $Q^{/,//,*,<}$ and $Q^{/,//,*,<,\neg}$, respectively. Based on these results, we have proposed a new holistic algorithm called TreeMatch to achieve such theoretical optimal query classes. Finally, extensive experiments demonstrate the advantage of our algorithms and verify the correctness of theoretical results.

Acknowledgments

This paper was partially supported by 863 National High-Tech Research Plan of China (No: 2009AA01Z133, 2009AA01Z149), National Science Foundation of China (NSFC) (No.60903056), Key Project in Ministry of Education (No: 109004) and SRFDP Fund for the Doctoral Program (No.20090004120002) and the Program for New Century Excellent Talents in University.

References

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

[2] A. Berglund, S. Boag, and D. Chamberlin. XML path language (XPath) 2.0. W3C Recommendation 23 January 2007 http://www.w3.org/TR/xpath20/.

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

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

[5] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan. Twig2stack: Bottom-up processing of generalized-tree-pattern queries over xml document. In *Proc. of VLDB Conference*, pages 19–30, 2006.

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

[7] S. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proc. of VLDB*, pages 263–274, 2002.

[8] B. Choi, M. Mahoui, and D. Wood. On the optimality of the holistic twig join algorithms. In *Proceeding of DEXA*, pages 28–37, 2003.

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

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

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

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

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

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

[15] J. Lu, T. W. Ling, Z. Bao, and C. Wang. Extended xml tree pattern matching: theories and algorithms. In *Technical Report*, 2010.

[16] J. Lu, T. W. Ling, C. Chan, and T. Chen. From region encoding to extended dewey: On efficient processing of xml twig pattern matching. In *VLDB*, pages 193–204, 2005.

[17] J. Lu, T. W. Ling, T. Yu, C. Li, and W. Ni. Efficient processing of ordered XML twig pattern matching. In *DEXA*, pages 300–309, 2005.

[18] M. Moro, Z. Vagena, and V. J. Tsotras. Tree-pattern queries on a lightweight XML processor. In *VLDB*, pages 205–216, 2005.

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

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

[21] M. Shalem and Z. Bar-Yossef. The space complexity of processing xml twig queries over indexed documents. In *ICDE*, 2008.

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

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

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

[25] W. Wang, H. Wang, H. Lu, H. Jiang, X. Lin, and J. Li. Efficient processing of XML path queries using the disk-based F&B index. In *VLDB*, pages 145–156, 2005.

[26] T. Yu, T. W. Ling, and J. Lu. Twigstacklistnot: A holistic twig join algorithm for twig query with not-predicates on xml data. In *DASFAA*, pages 249–263, 2006.

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

Jiaheng Lu is an associate professor at the Renmin University of China. He got PhD degree in National University of Singapore and his advisor is Prof. Tok Wang Ling. 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 serves as a program member in top conferences including SIGMOD and VLDB.

Tok Wang Ling received his PhD 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.

Zhifeng Bao received the BS degree from the Department of Computer Science, School of Computing, National University of Singapore in 2006. He is currently working toward the PhD degree in the Department of Computer Science, School of Computing, National University of Singapore. His research interests include XML structured query processing, XML keyword search and data stream analysis.

Chen Wang is a Staff Researcher at IBM Research - China. He received the B.S. (2003) and M.S. (2006) in computer science from Fudan University at Shanghai, China. From 2006, he worked at IBM Research - China in Beijing, China as researcher. His area of research is XML data management, machine learning, data mining, especially for graph data management and social network analysis.