Extended XML Tree Pattern Matching: Theories and Algorithms

S.V.Hemanth, S.Keerthi, D.Prasad

Dept. of CSE, Shaaz College of Engg & Tech, Hyderabad, AP, India

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 algorithms are efficient techniques 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 and XQuery) define more axes and functions such as negation function, order-based axis, and wildcards. In this paper, 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.

Keywords

Query Processing, XML/XSL/RDF, Algorithms, Tree Pattern

I. Introduction

As business and enterprises 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, Fig. 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 Fig. 1b, the query answers are nodes "C₁" and "C₂." Efficient matching of XML tree patterns has been widely considered as a core operation in XML query processing. In recent years, many methods [3-4, 9, 11, 13, 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 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 speed up performance using indexes [5, 11], 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.

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 paper, we call an XML tree pattern with negation function, wildcards, and/or order restriction as extended XML tree pattern. Fig. 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 "~". 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 nonbranching 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 paper 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" ([3, 12, 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 postprocessing approach has an obvious disadvantage: it outputs many matching elements of nonreturn nodes that are unnecessary for the final results. In this paper, we develop a new encoding method to record the mapping relationships and avoid outputting nonreturn nodes.

A. Main Results

In general, given an extended XML tree pattern query which may include P-C, A-D relationships, order restriction,

denoted as Q=;==;Ã; and 2) queries with P-C, A-D relationships, wildcards, and order restriction, denoted as Q=;==;Ã;<=; and 3) queries with P-C, A-D relationships, wildcards, order restriction, and negation function, denoted as Q=;==;Ã;<=;:. For each category, we identify the respective optimal query class.

The technical contributions of this paper 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 suboptimality 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 Fig. 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=;==;Ã,Q=;==;Ã;<=; and Q=;==;Ã;<=;:). 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.
II. Preliminaries

A. Modeling of XML Data and Extended Tree Pattern Query

An XML database D is usually modeled as a rooted, node-labeled tree (in this paper, 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 and negative) and (parent-child and ancestor-descendant). For example, in fig. 2(b), A;B is a positive parent-child edge and A;C is a negative parent-child edge. We use a symbol "." 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 figs. 2(c) and 2(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 fig. 2(a), (c) is the selected return node.

![Fig. 5: Mapping Relationship Between an Extended Tree Pattern and a Document Tree](image)

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:

1. Query node types (i.e., tag name) are satisfied by the corresponding database elements and wildcards "**" can match any single database element;
2. The positive edge relationships (including positive parent-child and positives ancestor-descendant edges) between query nodes are satisfied by the corresponding database elements;
3. The negative edge relationships (including negative parent-child and negative ancestor-descendant edges) are satisfied, that is, no corresponding database element pairs exist; and
4. 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.

B. 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], and extended Dewey scheme [16]. In this paper, 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 nonroot element u, label(u) = label(v) : n, where v is the nth 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(parent(e)) = "A", tag(grandparent(e)) = "B", and tag(grandparent(e)) = "C". In order to achieve this goal, paper [16] used an encoding scheme 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 paper. But for the purpose of understanding this paper 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.

C. Basic Properties of Algorithms

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

1. 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 TA 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.

2. 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 paper makes the contribution to propose algorithms to achieve this property for a
much larger class of queries with negation predicates, wildcards, and order restriction.

### III. Theoretical Analysis

In this section, we establish a theoretical framework about "matching cross" which demonstrates the intrinsic reason for the suboptimality of existing holistic algorithms. The purposes of our study are (1). to provide insight into the characteristics of the holistic algorithms, and thus promotes our understanding about their behaviors; and (2). to lead to novel algorithms that can guarantee a larger optimal query class and realize better query performance.

#### A. Matching Cross

The existing holistic algorithms [11, 16], consist of two phases: 1) 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 (2) 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) maybe produced in the first phase, which is called the suboptimality of algorithms, as further illustrated in the following example:

**Example 1.** Consider the document and query in fig. 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 $\delta A_1; B_1$), 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 $\delta A_i; B_i\beta$, which may become "useless" intermediate path solution if there were no $C_i$ 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 $B_i$ in $T_B$ is the first match of $A_i$ denoted as $FM(A_i, B_i)$, if only if $FM(A_i, B_i)$ and $FM(B_i, A_i)$ is a match binding, which contradicts that $B_i$ is the first match of $A_i$.

**Figure 6:** Illustration to Matching Cross

![Image](image6)

In this section, we establish a theoretical framework about "matching cross" which demonstrates the intrinsic reason for the suboptimality of existing holistic algorithms. The purposes of our study are (1). to provide insight into the characteristics of the holistic algorithms, and thus promotes our understanding about their behaviors; and (2). to lead to novel algorithms that can guarantee a larger optimal query class and realize better query performance.

**A. Matching Cross**

The existing holistic algorithms [11, 16], consist of two phases: 1) 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 (2) 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) maybe produced in the first phase, which is called the suboptimality of algorithms, as further illustrated in the following example:

**Example 1.** Consider the document and query in fig. 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 $\delta A_1; B_1$), 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 $\delta A_i; B_i\beta$, which may become "useless" intermediate path solution if there were no $C_i$ 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 $B_i$ in $T_B$ is the first match of $A_i$ denoted as $FM(A_i, B_i)$, if only if $FM(A_i, B_i)$ and $FM(B_i, A_i)$ is a match binding, which contradicts that $B_i$ is the first match of $A_i$.

**Figure 6:** Illustration to Matching Cross

![Image](image6)

In this section, we establish a theoretical framework about "matching cross" which demonstrates the intrinsic reason for the suboptimality of existing holistic algorithms. The purposes of our study are (1). to provide insight into the characteristics of the holistic algorithms, and thus promotes our understanding about their behaviors; and (2). to lead to novel algorithms that can guarantee a larger optimal query class and realize better query performance.

**A. Matching Cross**

The existing holistic algorithms [11, 16], consist of two phases: 1) 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 (2) 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) maybe produced in the first phase, which is called the suboptimality of algorithms, as further illustrated in the following example:

**Example 1.** Consider the document and query in fig. 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 $\delta A_1; B_1$), 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 $\delta A_i; B_i\beta$, which may become "useless" intermediate path solution if there were no $C_i$ 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 $B_i$ in $T_B$ is the first match of $A_i$ denoted as $FM(A_i, B_i)$, if only if $FM(A_i, B_i)$ and $FM(B_i, A_i)$ is a match binding, which contradicts that $B_i$ is the first match of $A_i$.

**Figure 6:** Illustration to Matching Cross

![Image](image6)
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 suboptimality of holistic algorithms, as illustrated follows:

Definition 3.3 (Bounded Matching Cross). Given a query Q and an XML database D, assume that \(<A; A; B_0; B_j>) is a matching cross for D with respect to Q. If the number of distinct elements \(A_m\) where, \(i \leq k \leq j\) and \(F \text{Mod} A_m; B_B \neq B_j < k_o\),

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 as no more than \(\text{HEIGHT}(D_j)\), we can buffer all such \(A_j\) in the main memory and read \(A_j\) to find the matching element for \(B_j\).

Example 2. Consider the query and document in fig. 8. \(<C_j; C_j; B_j; B_m \text{Mod} A_j>\) is a BMC, while \(<A; A; B_0; B_j>) is a UMC.

As shown in Definition 3.3 and Example 2, matching cross can be separated into 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 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 suboptimality 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 fig. 9 and an XPath query "\(H_j;==B; \neq A\)" (A is the selected return query node). \(<B_j; B_j; A_j; A_j>) is a UMC, since i and j maybe greater than the height of XML tree. But, we observe that this UMC still can be efficiently processed by registering the information that \(H_j\) has appropriate children B (e.g., \(B_j\)) and then scanning \(B_j\) and \(H_j\). Then, we can get an exact match (\(H_j; B_j; A_j\)) without outputting any possibly useless intermediate path. This example shows that the existence of UMC does not necessarily result in the suboptimality 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 the 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; A; B_0; B_j>) is a UMC on D with respect to Q, and A is a return node in Q but B is a nonreturn node. We call the node H 2 Q as a mediator node (H maybe a return node or not) if the first matches of all elements between \(B_0\) and \(B_j\) against node H are in the range from \(H_j\) to \(H_j\), and the first matches of all elements between \(H_j\) and \(H_j\) are between \(A_j\) and \(A_j\) (see fig. 10); and the number of elements between \(H_j\) and \(H_j\) that are the first matches of \(B_j\) and \(B_j\) is no more than the height of D.

For example, consider fig. 9 and the query "\(H_j;==B; \neq A\)" again. \(<B_j; B_j; A_j; A_j>) is a UMC, and H is a mediator in this UMC, as the first matches of all elements between \(B_j\) and \(B_j\) against node H is \(H_j\); and the first match of \(H_j\) and \(H_j\) are \(A_j\) and \(A_j\); and 2 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 fig. 9, we only need to buffer \(H_j\) and \(H_j\) to the main memory and record that \(H_j\) and \(H_j\) have the matching element with node B. Note that we do not need to buffer \(B_j\); \(B_j\); \(B_j\) 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.

For example, fig. 11, shows four example queries. \( Q_1, Q_2, Q_3 \) belong to mediator subclass, but \( Q_4 \) does not because 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 \( H_2 Q \) 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 (1) any concrete labeling schemes and (2) any special data index structures, such as XB tree [3], XR tree [11], and R tree [7]. This is because (1) the proof of the above theorems does not rely on any specific labeling scheme, and (2) 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.

**IV. 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, as the main bottleneck of optimality is the length of bitVector equals to the number of child nodes of \( q \). Given a node \( q \), 2 children \( \delta_q \), we use bitVector \( \varepsilon_q \) to denote the bit for \( q \). Specifically, bitVector \( \varepsilon_q \) is a query answer if and only if there is an element \( e_q \) in the document such that the \( e_q \) and \( e_q \) satisfy the query relationship between \( q \) and \( q \).

Finally, output list 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 \( S_q \), if all bits in bitVector \( \varepsilon_q \) 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 2) element \( e_q \) is in outputList \( \delta_q \) 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 \( e_q \) is in outputList \( \delta_q \) is a query answer can be accurately reflected by the corresponding bitVector \( \varepsilon_q \), illustrated as follows:

Example 3. Fig. 12, illustrates the set encoding \( S_q \) to query node \( A \) for an example document. There are two tuples in set \( S_q \). Since \( A_q ("0") \) has only one child \( B \) and no child element to match C, bitVector \( \varepsilon_q \) is a query answer in "10". In contrast, bitVector \( \varepsilon_q \) is not a query answer because of the P-C relationships in the query. Since all bits in bitVector \( \varepsilon_q \) are "1", \( B_q ("0.1.0") \) is guaranteed to be a query answer.

In our algorithm, we will frequently use the following two notations:

1) \( NAB(e) \) denotes the Nearest Ancestor Branching node of \( e \) in the query pattern \( Q \). Formally, \( q^e \) is a branching node and \( q^e \) is an ancestor of \( q \) and there is no other branching node \( q^{00} \) is in the path from \( q^e \) to \( q \). If there is no such ancestor of \( q \), then \( NAB(e) \) denotes the top branching node in query. 2) \( NDB(e) \) denotes the nearest descendants branching (or leaf) nodes of \( e \). Formally, \( q^e \) \( 2 NDB(e) \) if and only if \( q^e \) is a branching node and \( q^e \) is an ancestor of \( q \) and \( q^e \) is a descendant of \( q \) and there is no other branching or leaf node \( q^{00} \) is in the path from \( q^e \) to \( q \). For example, see the query \( Q_3 \) in fig. 11, \( NAB(D) \) \( ¼ FCg \), \( NAB(D) \) \( ¼ FBg \), \( NDB(D) \) \( ¼ FC; Dg \).

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.

---

**Fig. 11:** Example Queries to Illustrate Mediator Subclass. \( Q_1, Q_2, \) and \( Q_3 \) is in Mediator Subclass, But \( Q_4 \) Not. Because of (B; C) edge.

**Fig. 12:** Illustration to Set Encoding. (a) an XML Tree, and (b) a Query With Running-Time Set Encoding.
Example 4. Consider the data and query in Fig. 12 again. Note that B is the single-return node. First, B₁ and C₁ are read. Since A₁ now has only one child B₂ and one descendant C₂ (not child), we insert A₁ to set S₂ and bitVectorδA₁,B ¼ "10" (see Table 1). Next, when B₂ and C₂ are read, since A₂ has two children B₃ and C₃, we add A₂ to set and bitVectorδA₂,B ¼ "11". Finally, we empty set S₂ and output one element B₂ to the outputlist. 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 B₂, as bitVectorδA₂,B is "10". But TwigStack and TJFast would output two "useless" elements A₂ and B₂ in that case, and therefore, entail more I/O cost.

3. Tree Match
Now we go through Algorithm 1. Line 1 locates the first elements whose paths match the individual root-leaf path pattern. In each iteration, a leaf node fact is selected by getNext function (line 3). The purpose of lines 4 and 5 is to insert the potential matching elements to outputlist. Line 6 advances the list Tfact 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.

Algorithm 1. Algorithm TreeMatch for class Q=;==;Ã
1. locateMatchLabel(Q);
2. while (endRootP) do
3. fact ¼ getNextTopBranchingNodeP;
4. if (fact is a return node)
5. addT oOutputListNABðfactÞ; curTfactP;
6. advanceTfactP; // read the next element in Tfact
7. updateSetðfactÞ; // update set encoding
8. locateMatchLabelQP; // locate next element with matching path
9. emptyAllSetsrootP;

In Procedure addT oOutputListðq; ePÞ, we add the potential query answer eP to the set of S_{qi}, where, q is the nearest ancestor branching node of q (i.e., NABðqi,P ð" 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: 1) if n is a leaf node, f ¼ n (line 2); else 2) n is a branching node, then suppose element e matches node n in the corresponding path solution (if more than one element that matches n, e is the deepest one by level) (lines 7 and 8), we return f_{min} such that the current element emin in T_{min} has the minimal label in all ei by lexicographical order (lines 13 and 20).

Algorithm 2. Procedures and Functions in TreeMatch
(a) Procedure locateMatchLabel(Q) • for each leaf q 2 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; eP)
• for each e 2 S_q do
• if (e matches the individual root-leaf path)
• output ListðePÞ; addðeP; ePÞ;
Function satisfyTreePattern(eP, e)
• if (|bitVectorðeq; qP ð" 1") return true;
• else return false;
Procedure updateSet(q; e)
• cleanSetðq; eÞ; eP;
• add to set S_{e}; //set the proper bitVector(e)
• if (isRootðqÞ ^ (bitVectorðeq; qP ð" 1 1 . . . 1") update AncestorSetðeqÞ;
Procedure cleanSet(q; e)
• for each element e 2 S_q do
• if (e matches the individual root-leaf path)
• if (isRootðqÞ)
• addT oOutputListðNABðeqÞ; ePÞ;
• if (isTopBranchingðePÞ)
• if (there is only one element in S_{eq})
• output all elements in outputListðeqÞ;
• else merge all elements in outputListðeqÞ to outputListðeqÞ;
• delete e_q from set S_{eq};
Procedure updateAncestorSet(q)
• if (isRootðqÞ)
• for each e 2 S_q do
• if (e matches the individual root-leaf path)
• if (isRootðqÞ)
• delete e_q from set S_{eq};
Procedure emptyAllSets(q)  
• if (q is not a leaf node)  
  1. for each c of q do EmptyAllSets(c);  
• for each element e 2 S_q do cleanSet(q; e);  
Algorithm 3. getNext(n)  
1: if (isLeaf(n)) then  
2: return n  
3: else  
4: for each n 2 NDB(n) do  
5: f = ¼ getNext(n); b  
6: if (isBranching(n); emptyS_n b)  
7: return f  
8: else e ¼ maxfpjp 2 MB(n); nP  
9: end for  
10: max ¼ maxarg feg  
11: for each n 2 NDB(n) do  
12: if (8e 2 MB(n); nP : e 62 ancestors(emax))  
13: return f  
14: end if  
15: end for  
16: min ¼ minarg f if, is not a return nodeg  
17: for each e 2 MB(n_min; n)  
18: if (e 2 ancestorsde max; nP) updateSet(S_n; e)  
19: end for  
20: return f_min  
21: end if  
Function MB(n); b  
• if (isBranching(n)) then  
  1. Let e be the maximal element in set S_n  
  2. else  
  3. Let e ¼ ´1 curdTn  
  4. end if  
• 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  
For the second task of getNext, before an element e is inserted to the set S_n, we ensure that e is an ancestor (or parent) of another element e^ in S_n to match node b in the corresponding path solutions (line 13). If there are more than one element to match the branching node b, ebi is defined as their deepest (i.e., maximal) element (line 8).

Example 5. We use the query and document in fig. 13, to illustrate TreeMatch algorithm. Table 2 demonstrates the 4: current access elements, the sets encoding and the 5: corresponding output elements. There are two branching 6: if nodes in the query. First, B_2, D_1, and E_1 are scanned.  

----

Table 2: Set Encoding for the Example in fig. 13

<table>
<thead>
<tr>
<th>Current elements</th>
<th>Set encoding S_A</th>
<th>Set encoding S_C</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_1, D_1, E_1</td>
<td>&lt;0,1&gt;10^2,0&gt;</td>
<td>&lt;0,1&gt;10^2,0&gt;</td>
</tr>
<tr>
<td>B_1, D_1, E_2</td>
<td>&lt;0,1&gt;10^2,0&gt;</td>
<td>&lt;0,1&gt;10^2,0&gt;</td>
</tr>
<tr>
<td>B_2, D_1, E_2</td>
<td>&lt;0,1&gt;10^2,0&gt;</td>
<td>&lt;0,1&gt;10^2,0&gt;</td>
</tr>
</tbody>
</table>

---

Fig. 14: An Example of Ordered XML Tree Pattern Query. When we scan B_2 and C_1, we do not insert C_2 to minChild, as it is before B_2 and does not satisfy the order condition of query. (a) Order-based query. (b) Document.

C_1 and C_2 are added into the set S_n, 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_1 = C_1 = D_2 and A_2 = A_2 = C_2 = E_2, 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_n. 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 intermediate path A_1 = A_2 = C_1 = C_2 = E_2, 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).

**B. Extension for Order-Based Queries Q&lt;==;=;A;>&lt;**

In this section, we extend TreeMatch algorithm to support ordered-based queries Q&lt;==;=;A;>&lt;. 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 five-tuple: <label; bitV ector; outputList; 

---

Table 3: Partial Set Encoding for the Example in fig. 14

<table>
<thead>
<tr>
<th>Current elements</th>
<th>Set encoding S_A</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_1,C_1</td>
<td>(0.2,0)</td>
</tr>
<tr>
<td>B_1,C_2</td>
<td>(0.2,0.3)</td>
</tr>
<tr>
<td>B_2,C_3</td>
<td>(0.2,0.3)</td>
</tr>
</tbody>
</table>

that q_1, . . . , q_n are the children node of q (in order) in the query.
Given an element $e^{\ominus}$ in $\text{minChild}deq$ and $e^{\ominus\ominus}$ in $\text{maxChild}deq$, $e^{\ominus}$ is the minimal element that is greater than $q$, the element $e^{\ominus\ominus}$ is the maximal element that is smaller than $e^{\ominus\ominus}$ (if any). In particular, $e^{\ominus\ominus}$ is the leftmost children of $e$, and $e^{\ominus\ominus}$ is the rightmost children of $q$.

Example 6. See the query and document in fig. 14. Table 3, shows the values of $\text{minChild}$ and $\text{maxChild}$ attributes in set. (Note that the full presentation of each element in $S_q$ is a five-tuple. Here, we only show $\text{minChild}$ and $\text{maxChild}$ for the purpose of this example.) When $B_1$ and $C_2$ are read, since $C_1$ is before $B_2$, we do not insert $C_1$ as a minChild, as it is not greater than $B_2$. Only after $C_2$ is read, we insert $C_2$ to minChild. When $B_1$ and $C_2$ 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 $\text{satisfyTreePattern}$, we also need to check the order restriction according to minChild and maxChild.

Algorithm 4. Algorithm TreeMatch for class $Q=\{:,\};<$
1 Procedure updateSet($q$; $e$)
2: add $e$ to set $S_q$ //set the proper bitVector, negBitVector, minChild and maxChild

Function satisfyTreePattern($q$; $e$)
  • assume that child nodes of $q$ in $Q$ are $q_1; \ldots ; q_n$ (in order)
  • if ($e < \text{minChild}deq$; $q_\ominus\ominus$) return false;
  • if ($e > \text{maxChild}deq$; $q\ominus\ominus$) return false;
  • if ($\text{bitVector}deq$; $q_\ominus\ominus$ ("1") return true;
  • else return false;

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, these extra CPU cost is worthwhile.

C. Extension for Queries with Negative Edges $Q=\{:,\};=;<;$

The length of $\text{minChild}deq$ and $\text{maxChild}deq$ is equal to the number of children of $q$. Assume if and only if there is no element $eqc$ in the document such that the eq and $eqc$ satisfy the query relationship in between $q$ and $eq$. In this way, in order to know whether all negative children of $q$ are satisfied, we only check whether all children of $\text{negBitVectors}$ are "0". In line 2 of Procedure updateSet, we need to set the proper negBitVector according to the current elements. In Function satisfyTreePattern, $e$ is a valid element only if the negBitVector is "0".

Algorithm 5. Algorithm TreeMatch for class $Q=\{:,\};=;<;$
1. Procedure updateSet($q$; $e$)
2. add $e$ to set $S_q$; //set the proper bitV ector, negBitV ector, minChild and maxChild

Function satisfyTreePattern($q$; $e$)
  • if ($e < \text{minChild}deq$; $q_\ominus\ominus$) return false;
  • else if ($e > \text{maxChild}deq$; $q\ominus\ominus$) return false;
  • else if ($\text{bitVector}deq$; $q_\ominus\ominus$ "1") return true;
  • else return false;

D. Analysis of Algorithms

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

Lemma 4. In Algorithm $\text{TreeMatch}$, suppose any element $e_q$ is popped from set $S_q$, where, $q$ is the top branching node in Procedure $\text{CleanSet}(q)$, then $e_q$ matches the whole query if and only if all bits in bitVector $\text{ector}$ are "1" and all children of $e_q$ satisfy order condition (if any), and all bits in negBitVector $\text{ector}$ are "0" (if any).

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

Using Lemmas 4 and 5, we can see that whether or not an element is a query answer is exactly reflected by the values of the corresponding bitVector, negBitVector and $\text{minChild}$, $\text{maxChild}$. Further, by lines 5-7 in Procedure $\text{CleanSet}$, all correct solutions are output. In addition, each matching element is guaranteed to be inserted to the related sets in Procedure $\text{addToOutputList}$. 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 $\text{TreeMatch}$ correctly returns all the answers for $Q$ on $D$.

While the correctness holds for 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 $\text{CleanSet}$ belongs to final query solutions. Next, we show the corresponding optimality query subclass for three categories of queries, i.e., $Q=\{:,\};=;<;$, $Q=\{:,\};=;<;$, and $Q=\{:,\};=;<;<$.

Theorem 7. Consider an XML database $D$ and an extended tree pattern query $Q=\{:,\};<;\text{in mediator subclass}$ (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 $O(d^2Ab \& d\&X)$, 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 $b$ be a mediate node for any UMC involving in $q$. In Procedure $\text{addToOutputList}$, when each element $e_q$ is inserted to outputList, it is guaranteed to satisfy the subtree rooted with $q$ (line 2). In Procedure $\text{UpdateAncestorSet}$, the elements in outputList is moved to its ancestor set only if the current subtree is satisfied. We recursively guarantee that each eq in outputList satisfies the whole tree pattern. Therefore, each element eq is inserted to outputList of $e_q$ only if $e_q$ satisfies the whole tree pattern. We can safely write each element in outputList to disk in Procedure $\text{CleanSet}$ 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 maybe 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 ordered 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 $\frac{1}{2}$ 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 $Q_{\text{optimal}}$ 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 $O(2^{d}A b + d A f)$, 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., $Q_{\text{optimal}}$), 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 $Q_{\text{optimal}}$). We say that a query Q belongs to the optimal subclass for $Q_{\text{optimal}}$: 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 $\frac{1}{4}$ 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 $Q_{\text{optimal}}$: 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 $O(2^{d}A b + d A f)$, 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].

V. 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 TwigStack-ListNot [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) TwigStackNot is proposed for queries with negative edges, but it cannot work for ordered queries. Only TreeMatch algorithm can process queries with order restriction, negative edge, and wildcards.

A. Experiment Settings and Data Set

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 1.7 GHz CPU and 2 G of RAM. To offer a comprehensive evaluation of our new algorithms, we conducted experiments on both

Fig. 15: Queries for random data. (a) Q1 (optimal). (b) Q2 (optimal). (c) Q3 (suboptimal). (d) Q4 (optimal). (e) Q5 (optimal). (f) Q6 (suboptimal).

Fig. 16: Queries for DBLP (Q7-Q9) and TreeBank (Q10-Q13) data. (a) Q7 (optimal). (b) Q8 (suboptimal). (c) Q9 (optimal). (d) Q10 (optimal). (e) Q11 (suboptimal). (f) Q12 (optimal). (g) Q13 (optimal).

Fig. 16: Queries for DBLP (Q7-Q9) and TreeBank (Q10-Q13) data. (a) Q7 (optimal). (b) Q8 (suboptimal). (c) Q9 (optimal). (d) Q10 (optimal). (e) Q11 (suboptimal). (f) Q12 (optimal). (g) Q13 (optimal).

synthetic and real XML data. The synthetic data set is generated randomly. There are totally seven tags A, B, . . . , F , G in the data set 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 and maximal depth: 36). We can easily find queries on this data set to demonstrate the suboptimality for our tested algorithms.

B. Query Class $Q_{\text{optimal}}$ $A$

In this section, we show the experimental results for query class $Q_{\text{optimal}}$ $A$. All queries tested in our evaluation are shown in figs. 15 and 16.

1. 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 suboptimal for D3. This can be explained that D3 is a larger document than D1 and D2 so that D3 manifests the suboptimality which is hidden in D1 and D2. Fig. 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.

2. 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 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 and Q6 belong to the optimal query class. We compared the performance of three algorithms in figs. 17(b) and 18(a). Obviously, TreeMatch is superior to TwigStack and TJFast, reaching 20 to 95 percent improvement in execution time for all queries.

3. Medium Size of Main Memory
In most real applications, 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 fig. 19. In this experiment, we choose Q1 and Q6, since Q1 is an optimal query for TreeMatch, but Q6 is suboptimal. 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.

C. Query Class Q=;==;Ã<;:
In this section, we show the experimental results for query class Q=;==;Ã<;: which may contain order restriction, negative edge, and wildcards. The tested queries are shown in figs. 20 and 21. Table 6 shows the number of output elements and 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 fig. 21. Since Q17, Q18, Q23, and Q24 have negative edges, we compare TreeMatch with TwigStackList-Not [26] in fig. 18(b). In addition, as Q14-Q16 and Q20-Q22 are order-based queries, we compare TreeMatch with OrderedTJ [17] in Fig. 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

Table 4: Number of Output Elements (O) and the Percentage (P) of Useful Elements for TreeMatch on Random Data

<table>
<thead>
<tr>
<th>Query</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>132</td>
<td>100%</td>
<td>132</td>
</tr>
<tr>
<td>Q2</td>
<td>355</td>
<td>100%</td>
<td>177</td>
</tr>
<tr>
<td>Q3</td>
<td>955</td>
<td>98.3%</td>
<td>952</td>
</tr>
<tr>
<td>Q4</td>
<td>605</td>
<td>100%</td>
<td>365</td>
</tr>
<tr>
<td>Q5</td>
<td>291</td>
<td>100%</td>
<td>131</td>
</tr>
<tr>
<td>Q6</td>
<td>755</td>
<td>100%</td>
<td>94</td>
</tr>
</tbody>
</table>

Fig. 17: Execution time of Q=;==;Ã on random data. (a) Small memory. (b) Large memory.

Table 5: Number of Required Buffered Elements (Random Data)

<table>
<thead>
<tr>
<th>Query</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Q2</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Q3</td>
<td>528</td>
<td>27067</td>
<td>89779</td>
</tr>
<tr>
<td>Q4</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Q5</td>
<td>7</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>Q6</td>
<td>520</td>
<td>26808</td>
<td>89627</td>
</tr>
</tbody>
</table>

Fig. 18: Execution time on DBLP, TreeBank data (large memory). (a) Q=;==;Ã. (b) Q=;==;Ã<;:

www.ijcst.com
performance is achieved by using a relatively very small buffer size (see Table 7), we expect that our system can scale well for even gigabytes of XML data based on the current machine.

VI. Related Work

In the context of semistructured and XML databases, treepattern 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 [18-19, 22, 27]. 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 and Bar-Yossef [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 $Q=\text{select} \times \text{filter}$. Our research, in this paper, moves the frontier forward by identifying a large subclass of $Q=\text{select} \times \text{filter}$, which can be guaranteed to process optimally.

Table 6: Number of Output Elements and the Percentage of Useful Elements for TreeMatch on Random Data

Table 7: Number of Required Buffered Elements (Random Data)
The recent papers [5-6, 17, 26] are also closely related to ours. In [17], a new holistic algorithm, called OrderedTJ, is proposed to process order-based XML tree query. In [26], an algorithm called TwigStack-ListNot is proposed to handle queries with negation based query function. Note that the optimal query classes identified in those papers are smaller than that in this paper. 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, Twig2Stack [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 state- ments. But extended XML tree pattern is defined to include some complicated conditions like negative function, wild- card, and order restriction.

Besides the holistic algorithms, there are other approaches to match an XML tree pattern, such as ViST [23-24] and PRIX ([20]), which transform an XML tree pattern match to sequence match. Their algorithms mainly focus on ordered queries, and it is nontrivial to extend those methods to handle unordered queries and extended queries studied in this paper. 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 guaran- tees, is the most robust approach. In this paper, 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.

VII. Conclusions

We have introduced a notion of matching cross to address the problem of the suboptimality in holistic XML tree pattern matching algorithms. We have identified a large optimal query classes for three kinds of queries, that is Q=;==;Ã, Q=;==;Ã;<, and Q=;==;Ã;<;:; 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.

References


