Efficient Streaming Algorithms for Tree Matching Problems

Yangjun Chen\textsuperscript{1}, Leping Zou\textsuperscript{2}

Department of Applied Computer Science,
University of Winnipeg, Winnipeg, Manitoba, Canada R3B 2E9
\textsuperscript{1}yehn2@uwinnipeg.ca; \textsuperscript{2}zlpghost@gmail.com

Abstract- With the growing importance of XML in data exchange, much research has been done in providing flexible query mechanisms to extract data from XML documents. In this paper, we focus on the query evaluation in an XML streaming environment, in which data streams arrive continuously and queries have to be evaluated even before all the data of an XML document are available. Two algorithms will be discussed. One is for the unordered tree matching, by which only ancestor-descendant and parent-child relationships are considered. It requires \(O(|T| \cdot \text{leaf}_Q)\) time, where \(T\) is a subtree of document tree \(T\), in which each node matches at least one node in query \(Q\) and \(\text{leaf}_Q\) is the number of leaf nodes in \(Q\). The other is for the ordered tree matching, by which the left-to-right order of nodes must also be taken into account. It runs in \(O(|T| \cdot |Q|)\) time. Furthermore, our algorithms achieve high time performance without trading off space requirements. They have the same caching space and buffering space overhead as state-of-the-art stream- querying algorithm. We show the efficiency and effectiveness of our algorithms by a lot of experiments.

Keywords- XML Documents; Trees; Paths; XML Streams; XML Pattern Matching

I. INTRODUCTION

There is much current interest in processing streaming XML data, using queries expressed in languages such as XPath [52] and XQuery [53]. A streaming environment, as found with stock market data, network monitoring, or sensor network, differs from non-streaming XPath query processing in the following aspect. In a streaming environment, data streams which can be potentially infinite, arrive continuously, and must be processed in a single sequential scan due to the limited storage space available. Query results should be distributed incrementally once they are found, possibly before we have read all the data. In addition, the query processing algorithm should scale well in both time and space. An algorithm that meets such an environment for query evaluation over XML data is called a streaming evaluation algorithm.

In this paper, we address this issue, and propose two streaming algorithms for evaluating unordered and ordered tree matching, respectively.

- Data model and query language

Abstractly, an XML document can be considered as a tree structure with each node standing for an element name from a finite alphabet \(\Sigma\); and an edge for the element-subelement relationship.

In an XML streaming environment, an XML document tree \(T\) is modeled as a stream \(S\) of modified SAX (Simple API for XML) events: \texttt{startElement(tag, level, id)} and \texttt{endElement(tag, level)}, where \(tag\) is the tag of the node being processed, \(level\) is the level at which the node appears, and \(id\) is the unique identifier assigned to the node. A node in \(T\) exactly corresponds to a \texttt{startElement} (and the corresponding \texttt{endElement} event) in \(S\). In addition, if an element \(e\) has no subelement, a text is possibly associated with its \texttt{startElement}.

These events are the input to our query evaluation processor.

On the other hand, queries in XML query languages, such as XPath [52], XQuery [53], XML-QL [21], and Quilt [13, 14], typically specify patterns of selection predicates on multiple elements that also have some specified tree structured relations. For instance, the following XPath expression:

\begin{equation}
\text{book[title = 'Art of Programming']//author[fn = 'Donald' and ln = 'Knuth']}
\end{equation}

matches author elements that (i) have a child subelement \texttt{fn} with content 'Donald', (ii) have a child subelement \texttt{ln} with content 'Knuth', and are descendants of book elements that have a child \texttt{title} subelement with content 'Art of Programming'. This expression can be represented as a tree structure as shown in Fig. 1.

![Fig. 1 A query tree](image-url)

In this tree structure, a node \(v\) is labeled with an element name or a string value, denoted as \texttt{label}(v). Besides, there are two kinds of edges: child edges (\(c\)-edges) for parent-child relationships, and descendant edges (\(d\)-edges) for ancestor-descendant relationships. A \(c\)-edge from node \(v\) to node \(u\) is denoted by \(v \rightarrow u\) in the text, and represented by a single arc; \(u\) is called a \(c\)-child of \(v\). A \(d\)-edge is denoted \(v \Rightarrow u\) in the text, and represented by a double arc; \(u\) is called a \(d\)-child of \(v\). Also, a node in \(Q\) can be a wildcard '*' that matches any element in \(T\). Such a query is often called a twig pattern. In the following discussion, we use \texttt{startElement} and node interchangeably since each \texttt{startElement} event in \(S\) exactly corresponds to a node in \(T\).

- XML query evaluation and tree matching

- XML query evaluation and tree matching
In any DAG (directed acyclic graph), a node $u$ is said to be a descendant of a node $v$ if there exists a path (sequence of edges) from $v$ to $u$. In the case of a twig pattern, this path could consist of any sequence of $c$-edges and/or $d$-edges. Based on these concepts, the tree embedding can be defined as follows.

**Definition 1** (unordered tree matching) An embedding of a tree pattern $Q$ into an XML document $T$ is a mapping $f: Q \rightarrow T$, from the nodes of $Q$ to the nodes of $T$, which satisfies the following conditions:

(i) Preserve node label: For each $u \in Q$, $\text{label}(u) = \text{label}(f(u))$.

(ii) Preserve $c/d$-child relationships: If $u \rightarrow v$ in $Q$, then $f(v)$ is a child of $f(u)$ in $T$; if $u \Rightarrow v$ in $Q$, then $f(v)$ is a descendant of $f(u)$ in $T$.

If there exists a mapping from $Q$ into $T$, we say, $Q$ can be embedded into $T$, or say, $T$ contains $Q$. The purpose of XML query evaluation is to find all the subtrees of $T$, which contain $Q$.

Notice that an embedding could map several nodes with the same tag name in a query to the same node in a database. It also allows a tree mapped to a path, by which the order of siblings is totally unconsidered. This definition is a little bit different from the ordered tree matching defined below.

**Definition 2** (ordered tree matching) An embedding of a tree pattern $Q$ into an XML document $T$ is a mapping $f: Q \rightarrow T$, from the nodes of $Q$ to the nodes of $T$, which satisfies the following conditions:

(i) same as (i) in Definition 1.

(ii) same as (ii) in Definition 1.

(iii) Preserve left-to-right order: For any two nodes $v_1 \in Q$ and $v_2 \in Q$, if $v_1$ is to the left of $v_2$, then $f(v_1)$ is to the left of $f(v_2)$ in $T$.

$v_1$ is said to be to the left of $v_2$ if they are not related by the ancestor-descendant relationship and $v_2$ follows $v_1$.

We illustrate the left-to-right relationship in Fig. 2.

![Fig. 2 Illustration for left and right relatives](image)

This kind of tree mappings is useful in practice. For example, an XML data model was proposed by Catherine and Bird for representing interlinear text for linguistic applications, used to demonstrate various linguistic principles in different languages. For the purpose of linguistic analysis, it is essential to preserve the linear order between the words in a text. In addition to interlinear text, the syntactic structure of textual data should be considered, which breaks a sentence into syntactic units such as noun clauses, verb phrases, adjectives, and so on. These are used by the language TreeBank to provide a hierarchical representation of sentences. Therefore, by the evaluation of a twig pattern query against the TreeBank, the order between siblings should be considered.

For streaming algorithms, we distinguish between two kinds of usage-based memory space overheads:

- **Caching space** – the memory space used for the run-time stack to accommodate document elements.

- **Buffering space** – the memory space used to store potential answer nodes.

We denote the size of the caching and the buffering space by $CS$ and $BS$, respectively. Especially, $BS$ is measured as the maximal number of potential answer nodes buffered at any time point during the running time. If in $Q$ the nodes are labeled with different tag names, $BS$ is in the order of $|T|$. See [23]). However, if different nodes in $Q$ are labeled with the same tag name, $BS$ might be bounded by $O(|T|\cdot |Q|)$. In Fig. 3, we show a worst case.

![Fig. 3 Illustration for the worst case $BS$](image)

In the figure, querying $Q$ (shown in Fig. 3(a)) against $T$ (shown in Fig. 3(b)) has to buffer $c_1$ through $c_n$ for each $C$ in $Q$ until $b$ arrives. Fig. 3(c) is the XML document represented by $T$.

We will present two streaming algorithms according to the above two different definitions, respectively. The algorithm for evaluating unordered tree matching requires $O(|T|\cdot \text{leaf}(Q))$ time and $O(|Q|\cdot r)$ caching space, where $T'$ is a subtree of $T$, in which each node matches at least one node in $Q$ and $\text{leaf}(T') / (\text{leaf}(Q))$ represents the number of the leaf nodes of $T'$ (resp. $Q$), and $r$ is the maximal number of the nodes on a path in $T$, which are labeled with the same tag name. The algorithm for evaluating ordered tree matching runs in $O(|T|\cdot |Q|)$ time and $O(|Q|\cdot r)$ caching space.

The remainder of the paper is organized as follows. In Section II, we review the related work. In Section III, we discuss an algorithm for evaluating unordered tree matching. In Section IV, we discuss an algorithm for evaluating ordered tree matching. Section V is devoted to the experiments. Finally, a short conclusion is set forth in Section VI.

**II. RELATED WORK**

In the past two decades, there is much research on the tree pattern matching for the evaluation of XML queries. Nearly all the proposed strategies can roughly be divided into two categories. One is for the so-called XML streaming environment and the other is index-based.

- **Query evaluation in XML streaming environment**

  - Algorithm for evaluating unordered tree matching.
  - Algorithm for evaluating ordered tree matching.
A great many strategies have been proposed to evaluate XPath queries in an XML streaming environment [2, 12, 18, 23, 27, 32, 33, 39, 40]. The methods discussed in [2, 27] are based on finite state automata (FSA), but only able to handle single path queries, i.e., a query containing branching cannot be processed, as observed in [39]. The method proposed in [40] is a general strategy, but requires exponential time \( O(|T|^2) \) in the worst case, as analyzed in [40]. The methods discussed in [32, 33] do not support //edges. If we extend them to general cases, exponential time is required. In [12], Chen et al. first proposed a polynomial time algorithm, called TwigM, which runs in \( O(T_2|Q|T + |Q|^2 : |T|) \) time and \( O(|Q| : r) \) caching space, where \( T_2 \) is the height of \( T \) and \( Q_2 \) is the largest outdegree of a node in \( Q \). By this method, each node \( q \) of \( Q \) is associated with a boolean array of length \( q \)’s parent \( q’ \) satisfies the relationship between \( q \) and \( q’ \). Therefore, each time to figure out a stack and push a node into it, \( O(T_2|Q|T) \) time is required, leading to a time complexity of \( O(T_2|Q|T + |Q|^2 : |T|) \). See Theorem 4.4 in [12]. The algorithm discussed in [23] greatly improves TwigM, running in \( O(|Q| : r) \) time and \( O(|Q| : r) \) caching space.

- Query evaluation in indexing environment

Besides the above mentioned streaming, there is bunch of work on the index-based strategies. They can be further divided into two groups: the unordered tree pattern queries [5, 13, 14, 16, 19, 20, 24, 29, 34] and the ordered tree pattern matching [15, 43, 50, 51]. In the following, we only review some of them.

The method proposed in [34] is called XISS. It is a typical method based on indexing. By this method, single elements/attributes are indexed as the basic unit of queries and a complex path expression is decomposed into a set of basic path expressions. Atom expressions (single elements or attributes) are then recognized by directly accessing the index structure. However, all other kinds of expressions need join operations to stitch individual components together to get the final results.

Paths are also used as the basic indexing units by some methods, such as DataGuide [24] and Fabric [20]. By DataGuide, a concise summary of path structures for a semi-structured database is established, but restricted to raw paths. Therefore, complex path expressions or regular expression queries cannot be handled. Fabric works better in the sense that the so-called refined paths are supported. Such queries may contain branches, wild-cards and ancestor-descendant operators (/). However, any query not in the set of refined paths has to resort to join operations.

APEX improves DataGuide by introducing the so-called adaptive path indexes and uses data mining technique to summarize paths that frequently appear in the query workload. However, it suffers from two serious problems. First, it has to be updated as the query workload changes. Second, it keeps every path segment of length 2, instead of maintaining all paths starting from the root. Thus, to get the final results, the join operations have to be conducted. \( F'B \) [29] shares the flavour of Fabric [20]. It is based on the so-called forward and backward index (F&B index) [1], which covers all the branching paths. It works well for pre-defined query types. In normal cases, however, such a set of F&B indexes tends to be large and therefore the performance degrades.

By all the above methods, the indexes over binary relationships between pairs of nodes, such as parent-child and ancestor-descendant relations, or the indexes over paths can be very large, even when the input and final result sizes are much more manageable. As an important improvement, TwigStack was proposed by Bruno et al. [3], which stores the intermediate results in stacks to represent in linear space a potentially exponential number of answers. However, TwigStack are suitable only for the queries that contain no //edges. If a query contains both //edges and //edges, some useless path matchings have to be conducted. In the worst case, \( O(|D| : Q) \) time is needed for doing the merge joins as pointed out by Chen et al. (see page 287 in [19]). This method is further improved by several researchers, such as iTwigJoin [9] which exploits different data partition strategies, and TJFast [35] which accesses only leaf nodes by using extended Dewey IDs. By both methods, however, the path joins can not be avoided. In [19], Chen et al. discussed a method, calledTwigStack, which works in a quite different way. It represents the twig results using the so-called hierarchical stack encoding to avoid any possible useless path matchings. In [19], it is claimed that Twig2Stack needs only \( O(|D| : Q + |subTwigResults|) \) time for generating paths. A careful analysis, however, shows that the time complexity for this task is actually bounded by \( O(|D| : Q^2 + |subTwigResults|) \) for the following reason. Each time a node is inserted into a stack associated with a node in \( Q \), not only the position of this node in a tree within that stack has to be determined, but a link from this node to a node in some other stack has to be constructed, which requires to search all the other stacks in the worst case. The number of these stacks is \( |Q| \) (see Fig. 4 in [19] to know the working process).

In 2003, Wang et al. [50] proposed a first index-based method, called ViST, for handling ordered twig pattern queries. By this method, a document is stored as a sequence: \( (a_1, p_1), \ldots, (a_n, p_n) \), where each \( a_i \) is an element or a word in the document, and \( p_i \) a path from the root to it. Using this method, the join operations are replaced by searching a trie structure (wrongly called suffix tree in [50]). The drawback of this method is that a large index structure has to be created. (As pointed out in [43], the size of indexes can be higher than linear in the total number of elements in an XML document.) Another problem of this method is that a document tree that does not contain a query pattern may be designated as one of the answers due to the ambiguity caused by identical sibling nodes. This problem is removed by the so-called forward prefix checking discussed in [51].

Doing so, however, the theoretical time complexity is dramatically increased.

Such problems are removed by a method, called PRIX, discussed in [43]. This method constructs two Prifer sequences to represent an XML document: a numbered
The above algorithm processes the events in \( S \) one by one. Therefore, the corresponding document tree \( T \) is searched in the depth-first traversal fashion. Each time a startElement event is encountered, it will be pushed into
stack (see line 1 and lines 6 - 9) and stay there until its corresponding endElement is encountered (see lines 11 - 12). In this case, it will be popped out of stack and a node v for it will be constructed (see line 13), for which a containment check will be performed (see line 15).

Example 1 Consider the document tree T in Fig. 5(a). Its XML stream $S$ is shown in Fig. 5(b). Applying the algorithm query-evaluation(*) to $S$, we will regain $T$ if line 15 is not executed. In Fig. 6, we trace the first 9 steps of the execution process.

![Fig. 5 A document tree and its XML stream](image_url)

In the sample trace, special attention should be paid to Step 4, 8 and 9.

In Step 4, endE(c, 2) is encountered and the top element $E = (c, 2, 3)$ of the stack is popped out. For this element, a node $v_3$ is created. At the same time, a child link (from $E' = (a, 1, 2)$) is set to point to it since $E.P = 1$, which is the position (in the stack) where $E'$ is stored.

In step 8, node $v_4$ is constructed. It is a child of the element $(a, 1, 2)$ and therefore is appended to the end of the linked list associated with $(a, 1, 2)$. We also notice that $v_4$ itself is the parent of $v_5$.

In step 9, $v_2$ is generated. Since it is a child of $(a, 0, 1)$, a child link (from $(a, 0, 1)$) is established to point to it. Particularly, we have a subtree rooted at $v_2$ created.

![Fig. 6 Trace of Example 1](image_url)

From the above discussion, we can see that a document tree can always be constructed by scanning the corresponding XML stream $S$. For the purpose of query evaluation, however, we have to check the containment each time a node of $T$ is constructed. This is done by calling Algorithm containment-check(v, Q), in which another two functions are invoked to do different checkings:

- element-check(u, q): u is an element containing sub-elements. It checks whether $T[u]$ contains $Q[q]$. If it is the case, return $\{q\}$. Otherwise, it returns an empty set $\emptyset$.
- bottom-element-check(u, Q): u is an element containing no subelement. It returns a set of nodes in $Q$: $\{q_1, ..., q_k\}$ such that for each $q_i (1 \leq i \leq k)$ the following conditions are satisfied.
  (i) $label(u) = label(q_i)$.
  (ii) if $q_i$ has a child, then the child must be a text and matches the text associated with $u$.

**Algorithm containment-check(v, Q)**

**input:** v - a node in $T$; $Q$ - a twig pattern.

**output:** $\alpha(v)$ - a set of query node q such that $T[v]$ contains $Q[q]$.

**begin**
1. $C := \emptyset$; $C_1 := \emptyset$; $C_2 := \emptyset$; $\alpha := \emptyset$;
2. if v.c is not nil then (*v has some subelements.*)
3. \{ let $v_1, ..., v_k$ be the child nodes of $v$;
4. $\alpha := \alpha(v_1) \cup ... \cup \alpha(v_k)$;
5. for each $q \in \alpha$ do
6. \{ $\delta(q) := v$; $C := C \cup \{q\}$’s parent\};
7. remove all $\alpha(v_j) (j = 1, ..., k)$;
8. for each $q$ in $C$ do
9. $C_1 := C_1 \cup$ element-check(v,q);
10. \}
11. $C_2 :=$ bottom-element-check($v$, Q);
12. return $\alpha(v) = \alpha \cup C_1 \cup C_2$;
**end**

**Function element-check(u, q)**

**begin**
1. $C_1 := \emptyset$;
2. if label(q) = label(u) then (*If $q$ is '*'; the checking is always successful.*)
3. \{ let $q_1, ..., q_k$ be the child nodes of $q$;
4. if for each $q_i (i = 1, ..., k)$ $\delta(q_i)$ is equal to $u$
5. then $\{C_1 := \{q\}\}$;
6. if \( q \) is root then report \( u \);}
7. return \( C_i \);
end

Function bottom-element-check(\( u, Q \))
begin
1. \( C_z := \emptyset; \) flag := false;
2. for each leaf node \( q \) in \( Q \) do {
3. if \( q \) is a text then {
4. let \( q' \) be the parent of \( q \);
5. if label(\( q \)) = label(\( u \)) and \( q \) matches the text
associated with \( u \)
6. then \{ \( C_z := C_z \cup \{ q' \}; \) flag := true; \}
7. else {
8. if label(\( q \)) = label(\( u \)) then {
9. \( C_z := C_z \cup \{ q' \}; \) flag := true;
10. }
11. if \( q \) is root and \( \text{flag} := \text{true} \) then report \( u \);
12. flag := false;
13. }
14. return \( C_z \);
end

One of the inputs to the algorithm \( \text{containment-check}(v) \) is a node \( v \) constructed in the execution of query-evaluation(\( S, Q \)). If \( v \) corresponds to an element that has no subelement, the function \( \text{bottom-element-check}(v) \) is called (see line 11), by which \( \alpha(v) \) will be established by checking it against all the leaf nodes of \( Q \). Otherwise, \( \alpha(v_i) \) will be checked for all the child nodes \( v_i \) of \( v \) (see lines 3 - 6). Concretely, for each \( q \) in \( \alpha = \langle \alpha(v_1) \cup \ldots \cup \alpha(v_k) \rangle \), the value of \( \delta(q) \) will be changed to \( v \). Meanwhile, \( q \)’s parent will be stored in a temporary variable \( C \). Then, all the nodes \( q' \) in \( C \) are the candidates to be further checked. This is done by calling \( \text{element-check}(v, q') \) to see whether \( \mathcal{T}[v] \) contains \( Q[q'] \) (see lines 8 - 9). Special attention should be paid to the fact that \( \text{bottom-element-check}(v) \) should also be applied to \( v \) to find all the leaf nodes of \( Q \) which matches \( v \).

Finally, we notice that in the execution of \( \text{element-check}(v) \), \( \delta(q) \)'s are utilized to facilitate the checkings (see lines 3 - 5 in \( \text{element-check}(v) \)).

The following example helps for illustration.

Example 2 Consider \( T \) and \( S \) shown in Fig. 4(a) and \( Q \) shown in Fig. 7.

By executing query-evaluation(\( S, Q \)), the nodes of \( T \) will be constructed bottom up.

The first constructed node is \( v_3 \), by which \( \text{containment-check}(v_3, Q) \) is invoked. Since it is a leaf node and matches \( q_3 \) in \( Q \), \( \text{containment-check}(v_3, Q) \) returns \( \alpha(v_3) = \{ q_3 \} \) (see lines 11 in \( \text{containment-check}(v) \)).

The second constructed node is \( v_5 \). In the same way, we will set \( \alpha(v_5) = \{ q_2 \} \).

When \( v_4 \) is constructed, \( \text{containment-check}(v_4, Q) \) is called. Since it is the parent of \( v_4 \), we will first set \( \delta(q_2) \) to \( v_4 \) in terms of \( \alpha(v_4) = \{ q_2 \} \) (see line 4 and 6 in \( \text{containment-check}(v) \), and Fig. 7 for illustration.) After that, \( \text{element-check}(v_3, Q) \) is invoked. (Note that \( q_1 \) is the parent of \( q_2 \). See lines 8 - 9) Since \( \text{label}(v_1) = e \neq \text{label}(q_1) = a \), it returns \( C_1 = \emptyset \). \( \text{bottom-element-check}(v_2, Q) \) also returns \( C_2 = \emptyset \). So \( \alpha(v_4) = \alpha(v_3) \cup C_1 \cup C_2 = \{ q_1, q_2 \} \) (see line 12 in \( \text{containment-check}(v) \)).

When \( v_3 \) is constructed, \( \text{containment-check}(v_3, Q) \) is called. Since it is the parent of \( v_3 \), we will first set \( \delta(q_2) \) to \( v_4 \) in terms of \( \alpha(v_3) = \{ q_2 \} \) (see line 4 and 6 in \( \text{containment-check}(v) \), and Fig. 7 for illustration.) After that, \( \text{element-check}(v_3, Q) \) is invoked. (Note that \( q_1 \) is the parent of \( q_2 \). See lines 8 - 9) Since \( \text{label}(v_1) = e \neq \text{label}(q_1) = a \), it returns \( C_1 = \emptyset \). \( \text{bottom-element-check}(v_2, Q) \) also returns \( C_2 = \emptyset \). So \( \alpha(v_4) = \alpha(v_3) \cup C_1 \cup C_2 = \{ q_1, q_2 \} \) (see line 12 in \( \text{containment-check}(v) \)).

The next created node is \( v_2 \). For this node, we will first set \( \delta(q_2) = \delta(q_3) = v_2 \) in the execution of \( \text{containment-check}(v_2, Q) \) (in terms of \( \alpha(v_2) = \{ q_2 \} \) and \( \alpha(v_3) = \{ q_3 \} \), respectively). Next, we call \( \text{element-check}(v_2, q_3) \) to check whether \( \text{label}(v_2) = \text{label}(q_3) \). It is the case. So we will further check whether \( \delta(q_2) (i = 2, 3) \) is equal to \( v_2 \). Since both \( \delta(q_2) \) and \( \delta(q_3) \) are equal to \( v_2 \), we have that \( \mathcal{T}[v_2] \) contains \( Q[q_3] \). Therefore, \( \alpha(v_1) = \{ q_1 \} \), and \( \alpha(v_2) \) is set to be \( \alpha(v_3) \cup \alpha(v_2) \cup C_1 \cup C_2 = \alpha(v_3) \cup \alpha(v_4) \cup \{ q_1 \} \cup \emptyset = \{ q_1, q_2, q_3 \} \).

In a next step, \( v_1 \) will be constructed. It is a leaf node and matches \( q_2 \). Therefore, \( \alpha(v_1) = \{ q_2 \} \).

Similarly, we will set \( \alpha(v_3) = \{ q_3 \} \).

When \( v_3 \) is constructed, \( \delta(q_2) \) will be changed to \( v_5 \) (according to \( \alpha(v_3) = \{ q_3 \} \)), but \( \delta(q_3) \) remains not modified. Since \( \text{element-check}(v_3, q_i) \) returns \( \emptyset \), \( \alpha(v_3) = \alpha(v_2) \cup \alpha(v_1) \cup C_1 \cup C_2 = \{ q_1, q_2, q_3 \} \).

Finally, we will meet \( v_1 \). In the execution of \( \text{containment-check}(v_1, Q) \), \( \delta(q_1) \) is set to \( v_1 \), \( \delta(q_2) \) to \( v_1 \), and \( \delta(q_3) \) to \( v_1 \). Since \( \text{label}(v_1) = \text{label}(q_1) \), \( \delta(q_2) = v_1 \) and \( \delta(q_3) = v_1 \), \( \text{element-check}(v_1, q_i) \) returns \( \emptyset \). Therefore, \( \alpha(v_1) \) is set to \( \alpha(v_2) \cup \alpha(v_1) \cup C_1 \cup C_2 = \{ q_1, q_2, q_3 \} \).

B. Correctness and Complexities

In this subsection, we prove the correctness of \( \text{containment-check}(v) \) and analyze its computational complexities.

First of all, we denote by \( t(v) \) the time when \( v \) is created.

Concerning \( t(v) \), we have the following lemma.

Lemma 1 Let \( v_1, v_2, \) and \( v_3 \) be three nodes in a tree with \( t(v_1) > t(v_2) > t(v_3) \). If \( v_1 \) is a descendant of \( v_3 \), then \( v_2 \) must also be a descendant of \( v_3 \).

Proof. We consider two cases: i) \( v_2 \) is to the right of \( v_3 \); and ii) \( v_2 \) is an ancestor of \( v_1 \). In case (i), \( v_2 \) is pushed into \( stack \) later than \( v_1 \) and therefore later than \( v_3 \). This shows that \( v_2 \) is
containment of $\alpha$. Otherwise, if $v_2$ is not a descendant of $v_1$, we have $t(v_2) > t(v_1)$. Contradiction. In case (ii), $v_1, v_2,$ and $v_3$ are on the same path. Since $t(v_3) > t(v_2) > t(v_1)$, $v_2$ must be a descendant of $v_3$.

We illustrate Lemma 1 by Fig. 9.

![Fig. 9 Illustration for Lemma 1](image)

This lemma ensures that the value of $\delta(q)$ is always appropriately established. Although $\delta(q)$ may be changed more than once during the computation, at the time point we check $q$, $\delta(q)$ must be the parent or the ancestor of a node $v$ such that $T[v]$ embeds $Q[q]$.

From the above analysis, we have the following proposition.

**Proposition 1** Let $v$ be a node in $T$. Then, for each $q$ in $\alpha(v)$ generated by containment-check(), we have that $T[v]$ contains $Q[q]$.

**Proof.** The proposition can be proved by induction of the tree height by using Lemma 1.

Now we analyze the time complexity of the algorithm.

The dominant cost of the algorithm is the time spent on unifying $\alpha(v_i), ..., \alpha(v_k)$, where $v_i (i = 1, ..., k)$ is a child node of some node $v$ in $T$. This part of cost is bounded by

$$O(\sum_{v_i} d_i |Q|) = O(|T||Q|),$$

where $d_i$ represents the ouredge of a node $v_i$ in $T$.

However, this computational complexity can be improved by reducing the size of each $\alpha(v)$.

For this purpose, we assign each node $q$ in $Q$ a pair of numbers as follows. By traversing $Q$ in preorder, each node $q$ will obtain a number $s(q)$ when it is first encountered. When the search finishes examining all the children of $q$, it will get its second number $t(q)$. These two numbers can be used to characterize the ancestor-descendant relationships as follows.

Let $q$ and $q'$ be two nodes of a tree $Q$. Then, $q'$ is a descendant of $q$ iff $s(q') < s(q) < t(q) < t(q')$.

In addition, if $t(q) < s(q)$, $q'$ is to the left of $q$.

Assume that $q$ and $q'$ are two query nodes appearing in $\alpha(v)$. If $q'$ is a descendant of $q$, then we can remove $q'$ from $\alpha(v)$ since the containment of $Q[q]$ in $T[v]$ implies the containment of $Q[q']$ in $T[v]$. This can be done as follows.

First of all, we notice that the algorithm searches $T$ bottom-up. For a leaf node $v$ in $T$, $\alpha(v)$ is initialized with all those leaf nodes in $Q$, which match $v$. This can be carried out by searching the leaf nodes in $Q$ from left to right. Then, for any two leaf nodes $q$ and $q'$ in $\alpha(v)$, if $q'$ appears before $q$, we have that $s(q') < s(q)$. That is, $\alpha(v)$ is initially sorted by the $s(q)$ values. We can store $\alpha(v)$ as a linked list. Let $\alpha_1$ and $\alpha_2$ be two sorted lists with $|\alpha_1| \leq \text{leaf}_0$ and $|\alpha_2| \leq \text{leaf}_0$. The union of $\alpha_1$ and $\alpha_2$ ($\alpha_1 \cup \alpha_2$) can be performed by scanning both $\alpha_1$ and $\alpha_2$ from left to right and inserting the elements in $\alpha_2$ into $\alpha_1$ one by one. During this process, any element in $\alpha_1$, if it is a descendant of some element in $\alpha_2$, will be removed; and any element in $\alpha_2$, if it is a descendant of some element in $\alpha_1$, will not be inserted into $\alpha_1$. The result is stored in $\alpha_1$. Obviously, the resulting linked list is still sorted by $s(\cdot)$ values and its size is bounded by $\text{leaf}_0$.

We denote this process as $\text{merge}(\alpha_1, \alpha_2)$ and define $\text{merge}(\alpha_1, ..., \alpha_i, \alpha)$ to be $\text{merge}(\text{merge}(\alpha_1, ..., \alpha_i), \alpha)$. In this way, the time complexity of the algorithm can be improved to $O(|T|\cdot \text{leaf}_0)$.

In terms of the above analysis, we have the following proposition.

**Proposition 2** The time complexity of containment-check() is bounded by $O(|T|\cdot \text{leaf}_0)$.

**Proof.** See the above discussion.

Now we analyze the space complexity. First, we notice that at any time point stack contains only some nodes on a path in $T$. So the caching space must be bounded by $O(T_v)$. However, we can slightly change our algorithm to reduce the time complexity to $O(Q)$. Set.

1. In the execution of query-evaluation($S, Q$), each time we push an element $v$ into stack we will check whether the tag name of $v$ appears in $Q$. If it is the case, push $v$ into stack; otherwise, not. Using a hash table over an array containing all the tag names appearing in $Q$, this checking needs only $O(1)$ time.

2. In the linked list pointed to by $E, c$ (see Fig. 4) the nodes are the descendants of $E$, satisfying the following two conditions:

   i) All these nodes are not descendants of each other.

   ii) Let $v$ be one of these nodes. Then, on the path from $E$ to $v$, no node is labeled with a tag name appearing in $Q$.

In order to satisfy conditions (i) and (ii), we need to assign each node $v$ in $T$ two values: $s(v)$ and $t(v)$ as for the nodes in $Q$. $s(v)$ and $t(v)$ are generated when the corresponding startElement(tag, level, id) and startElement(tag, level, id) are encountered, respectively.

Finally, we point out that since the size of $\alpha(v)$ is bounded by $\text{leaf}_0$, the worst case buffering space can be reduced to $O(|T|\cdot \text{leaf}_0)$.

**C. General Cases**

The algorithm discussed in Subsection A can be easily extended to general cases that a query tree contains both /-edges and //edges, as well as wildcards and branches.

Let $q_1, ..., q_k$ be the child nodes of $q$. Let $v_1, ..., v_l$ be the
child nodes of \( v \). If \( T[v] \) contains \( Q[q] \), the following two conditions must hold:

- for each /-edge \((q, q_i)(1 \leq i \leq k)\), there must exist a \( v_j(1 \leq j \leq l)\) such that \((v, v_j)\) matches \((q, q_i)\), and
- \( T[v] \) contains \( Q[q] \).

In terms of this analysis, we modify Algorithm `containment-check` (as follows).

**Algorithm** `general-containment-check(v, Q)`

**Input:** \( v \) is a node in \( T \), \( Q \) is a twig pattern.

**Output:** \( \bigcirc (v) \) is a set of query node \( q \) such that \( T[v] \) contains \( Q[q] \).

**Begin**

1. \( C := \emptyset; \) \( C_1 := \emptyset; \) \( C_2 := \emptyset; \) \( \alpha := \emptyset; \)
2. if \( v, c \) is not nil then (**v** has some subelements.)*
3. \{ let \( v_1, \ldots, v_k \) be the child nodes of \( v \);
4. for \( i = 1 \) to \( k \) do { 
5. for \( q \in \alpha(v_i) \) do {
6. if \((q \in a/d\)-child) or
7. \( q \) is a c-child and \( q \) matches \( v_i)\))
8. \( \therefore \delta(q) := v \)
9. }
10. \( \alpha := \text{merge}(\alpha(v_1), \ldots, \alpha(v_k)); \)
11. assume that \( \alpha = \{q_1, \ldots, q_l\} \)
12. for \( i = 1 \) to \( j \) do {
13. if \((q_i \text{ is a } a/d\text{-child}) \) and \( \text{parent } \neq q_{i+1} \text{ parent})
14. then \( C := C \cup \{q_i \text{ parent}\}; \}
15. remove all \( \bigcirc (v_i) \) \( i = 1, \ldots, k \); 
16. for each \( q \) in \( C \) do 
17. \( C_1 := C_1 \cup \text{element-check}(v, q); \)
18. \} 
19. \( S_2 := \text{bottom-element-check}(v); \)
20. \( \alpha(v) := \text{merge}(\alpha, C_1, C_2); \)
**End**

The first difference of the above algorithm from the algorithm `containment-check` ( ) is that before we set the value for \( \delta(q) \) we will check whether \( q \) is a /-child or a /-child. If \( q \) is a /-child, we will further check whether it matches \( v_i \) (see lines 6 - 8). We notice that \( q \) appearing in \( \alpha(v_i) \) only indicates that \( Q[q] \) can be embedded into \( T[v] \), but not necessarily means that \( q \) matches \( v_i \).

The second difference is line 10 and lines 12 - 13. In line 10, we use the merge operation to union \( \alpha(v_1), \ldots, \alpha(v_k) \) together. In lines 12 - 13, we generate a set \( C \) that contains the parent nodes of all those nodes appearing in \( \alpha(\alpha(v_1), \ldots, \alpha(v_k)) \), where \( v_j \) is a child node of the current node \( v \). Since the nodes in \( \alpha \) are sorted (according to the nodes’ pre and post values), if there are more than one nodes in \( \alpha \) sharing the same parent, they must appear consecutively in the list. So each time we insert a parent node \( q' \) (of some \( q \) in \( \alpha \)) into \( C \), we need to check whether it is the same as the previously inserted one. If it is the case, \( q' \) will be ignored. Thus, the size of \( C \) is also bounded by O(\( |\text{leaf}_q| \)).

Fourth Algorithm for Ordered Tree Matching

In this section, we discuss our second algorithm according to Definition 2.

In the algorithm, we will keep two kinds of data structures defined below.

\( \chi(q) \) – a link associated with \( q \) in \( Q \), pointing to the leftmost leaf node in \( Q[q] \) as illustrated in Fig. 10(a).

![Fig. 10 Labeled trees and postorder numbering](image)

For a leaf node \( q' \); \( \chi(q') \) is defined to be \( q' \) itself. So in Fig. 9(a), we have \( \chi(q_1) = \chi(q_2) = \chi(q_3) = q_3 \). If we consider \( q' = \chi(q) \) as a function, we can also define its reverse function, denote by \( \chi^{-1}(q') \). Its value is a set containing all those nodes \( q \) such that \( \chi(q) = q' \), including \( q' \) itself. For example, for \( q_1 \) in Fig. 9(a), we have \( \chi^{-1}(q_3) = \{q_1, q_2, q_3\}, \chi^{-1}(q_3) = \{q_3\} \), and \( \chi^{-1}(q_3) = \{q_3\} \).

The nodes of \( Q \) are also numbered in postorder as shown in Fig. 9(b). So the nodes in \( Q \) will be referred to by their postorder numbers. In addition, a virtual node for \( Q \) is created, numbered 0 and considered to be to the left of any node in \( Q \).

\( B_v \) – an array of length \( |Q| \), indexed from 0 to \( |Q| - 1 \). In \( B_v \), each entry \( B_v[i] \) is a pointer to a unit containing a node \( j \) in \( Q \) such that \( j \) is the closest node to the right of \( i \) such that \( T[v] \) embeds \( Q[j] \). Initially, each \( B_v[i] \) is set to \( \phi \).

The unit pointed to by \( B_v[i] \) is denoted by \( f(B_v[i]) \).

Using \( B_v \), the ordered tree embedding can be checked as follows.

Let \( v \) be a node created by executing `query-evaluation` (\( S, Q \)). Let \( v_1, \ldots, v_k \) be the child nodes of \( v \). Let \( q_1, \ldots, q_k \) be the child nodes of \( q \) currently checked. To know whether \( T[v] \) embeds \( Q[q] \), we first check \( b_v \), starting from \( B_v[p] \), where \( p = \chi(q) - 1 \). We begin the searching from \( \chi(q) - 1 \) because it is the closest node to the left of the first child of \( q \).

Let \( f(B_v[p]) = p' \). If \( p' \) is an ancestor of \( q_1 \), this shows that \( T[v] \) embeds \( Q[q] \) and thus \( T[v] \) embeds \( Q[q] \).

If \( p' \) is \( q_1 \) and \( (q, q_1) \) is a /-edge, or both \( (q, q_1) \) and \( (v, v_1) \) are /-edges, we will check \( f(B_v[p']) \) in a next step to see whether \( f(B_v[p']) \) is an ancestor of \( q_2 \) or equal to \( q_2 \).

Otherwise, we check \( f(B_v[p]) \) in a next step to see whether \( f(B_v[p']) \) is an ancestor of \( q_1 \) or equal to \( q_1 \).

This process continues until one of the following conditions is satisfied:

(i) All \( B_v[i] \)’s \( (i = 1, \ldots, k) \) are exhausted, or
(ii) All \( q \) \((j = 1, \ldots, l) \) are covered.
Case (i) shows that $T[v]$ is not able to embed $Q[q]$ while the case (ii) indicates an embedding of $Q[q]$ in $T[v]$.

In the computation, $B_v$'s will dynamically be maintained as follow.

- If we find $Q[x]$ can be embedded in $T[v]$, we will create a unit $U$ containing $x$ and set $B_v[q_1], \ldots, B_v[q_s]$ to a be a pointer to $U$, where each $q_l (1 \leq l \leq s)$ is a query node to the left of $x$, to record the fact that $x$ is the closest node to the right of $q_l$ such that $T[v]$ embeds $Q[x]$.

- If some time later we find another node $x'$ such that $Q[x']$ can be embedded in $T[v]$, we distinguish between two cases:
  a) If $x'$ is to the right of $x$, we will generate a new unit $U'$ containing $x'$ and set $B_v[p_1], \ldots, B_v[p_s]$ to a pointer to $U'$, where each $p_l (1 \leq l \leq s)$ is to the left of $x'$ but to the right of $q_l$.
  b) If $x'$ is an ancestor of $x$, we will change $B_v$ as below.

Let $j' = \chi(x') + 1$. If $B_v[j]$ and $B_v[j + 1]$ point to different units, simply change the value in the unit pointed to by $B_v[j]$ to $x'$.

Otherwise, $B_v[j]$ and $B_v[j + 1]$ point to the same unit $W$. Create a new unit $W'$ and store the value of $W$ in $W'$. Find $k \geq j + 1$ such that $B_v[j + 1] = B_v[j + 2] = \ldots = B_v[k]$ but $B_v[k] \neq B_v[k + 1]$. Change all these entries to point to $W'$ and then change the value in $W'$ to $x'$.

We also note that once $B_v[j + 1], \ldots, B_v[k]$ are changed for the first time, they will not be changed any more. If we search $Q$ bottom-up. (That is, each time a node $v$ is created, we will search $Q$ bottom-up to find any $q$ such that $T[v]$ embeds $Q[q]$.) This property can be analyzed as follows.

Let $q'$ be a query node encountered in the subsequent computation such that $T[v]$ embeds $Q[q']$. Then, $q'$ must be to the right of $q$ or an ancestor of $q'$. If $q'$ is to the right of $q$, these entries obviously will not be changed. If $q'$ is an ancestor of $q$, we have $\chi(q') \leq j + 1$. If $Q[q'] = j + 1$, the unit pointed to by $B_v[j + 1]$ will be changed to $q'$. If $Q[q'] < j + 1$, some entries before $B_v[j + 1]$ will be changed. In both cases, $B_v[j + 1], \ldots, B_v[k]$ will not be touched. Therefore, for each $v$ in $T$, this part of cost is bounded by $|Q|$.

Finally, we need to merge each $B_v$ into $B_v$ since the embedding of a subtree in $T[v]$ implies the embedding of that subtree in $T[v]$. $merge(A_v, A_v)$ is defined as below:

$$merge(B_v, B_v, [j]) = \begin{cases} 
pointer to \max \{f(B_v[j]), f(B_v[j])\}, & \text{if } f(B_v[j]) \neq f(B_v[j]) \text{ and they are on the same path;} \\
pointer to \min \{f(B_v[j]), f(B_v[j])\}, & \text{otherwise.}
\end{cases}$$

In this definition, we handle $\phi$ as a negative integer (e.g., -1) and consider it as a descendant of any node. Obviously, if $f(B_v[j])$ and $f(B_v[j])$ are on the same path, $merge(B_v[j], B_v[j])$ should be set to be a pointer to $\max \{f(B_v[j]), f(B_v[j])\}$. (Here, we use $\max \{f(B_v[j]), f(B_v[j])\}$ to represent the unit containing $\max \{f(B_v[j]), f(B_v[j])\}$.) However, if $f(B_v[j])$ and $f(B_v[j])$ are on different paths, $merge(B_v[j], B_v[j])$ is set to be $\min \{f(B_v[j]), f(B_v[j])\}$. It is because in $B_v$ each entry $B_v[j]$ is a pointer to a unit containing the closest node $j'$ to the right of $j$ such that $T[v]$ contains $Q[j']$. After this operation, we can remove $B_v, \ldots, B_v$ since they will not be used any more.

From the above discussion, we can see that for each created node $v$, we need $O(d_v \cdot |Q|)$ time to check the tree embedding, where $d_v$ is the outdegree of node $v$ in $T$. So the total time complexity is bounded by

$$O(\sum_{i=1}^{T} d_v \cdot \min \{r_v, Q'\}) = O(|T| \cdot |Q|).$$

However, the buffering space is in the order of $O(\min \{r_v, Q'\})$. It is because after a $v$ is checked all the arrays associate with its children are removed. Thus, at any time point during the execution, at most $T_{leaf}$ nodes in $T$ are associated with a array.

The algorithm is somehow related to the method discussed in [30], in which each node in $Q$ is associated with an array of size $|Q|$. So its space complexity is in the order of $O(|T| \cdot |Q|)$. Especially, this method cannot be adapted to a data streaming environment.

V. EXPERIMENTS

In this section, we report the test results. We conducted our experiments on a DELL desktop PC equipped with Pentium(R) 4 CPU 2.80GHz, 0.99GB RAM and 20GB hard disk. The code was compiled using Microsoft Visual C++ compiler version 6.0, running standalone.

- Tested methods

In the experiments, we have tested four methods on the tree matching:
- TwigM [12],
- Gou's method [23],
- the method for the unordered tree matching (discussed in this paper, Chen-1 for short), and
- the method for the ordered tree matching (discussed in this paper, Chen-2 for short).

The theoretical computational complexities of these methods are summarized in Table 1.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Query Time</th>
<th>CS</th>
<th>BS</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwigM</td>
<td>$O(T \cdot \min {Q' \cap</td>
<td>Q</td>
<td>, r })$</td>
</tr>
<tr>
<td>Gou's</td>
<td>$O(</td>
<td>T</td>
<td>\cdot</td>
</tr>
<tr>
<td>Chen-1</td>
<td>$O(</td>
<td>T</td>
<td>\cdot</td>
</tr>
<tr>
<td>Chen-2</td>
<td>$O(</td>
<td>T</td>
<td>\cdot</td>
</tr>
</tbody>
</table>

- Data

The data sets used for the tests are TreeBank data set

\[ -9 - \]
In the experiments, we tested altogether 21 queries shown in Table 3, 4, and 5.

### TABLE III QUERIES FOR TREEBANK DATA SET

<table>
<thead>
<tr>
<th>query</th>
<th>XPath expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>//VP[DT]/PRP_DOLLAR</td>
</tr>
<tr>
<td>Q2</td>
<td>//S/VP/PP_IN]/NP</td>
</tr>
<tr>
<td>Q3</td>
<td>//S/VP/PP_IN]/NP/VB_IN</td>
</tr>
<tr>
<td>Q4</td>
<td>//VP/.//PP/IN]/NP/*//JJ</td>
</tr>
<tr>
<td>Q5</td>
<td>//S[CC]/.PP]/NP/VB_IN]/JJ</td>
</tr>
</tbody>
</table>

### TABLE IV QUERIES FOR DBLP

<table>
<thead>
<tr>
<th>query</th>
<th>XPath expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q6</td>
<td>//article_author=&quot;C.J. Codd&quot;</td>
</tr>
<tr>
<td>Q7</td>
<td>//inproceedings[author=&quot;Jim Gray&quot;][year=&quot;1990&quot;]</td>
</tr>
<tr>
<td>Q8</td>
<td>//inproceedings[key][author=&quot;Jim Gray&quot;][year=&quot;1990&quot;]</td>
</tr>
<tr>
<td>Q9</td>
<td>//inproceedings[title][.//pages][.//url]</td>
</tr>
<tr>
<td>Q10</td>
<td>//articles[author][title][.//volume][.//pages][.//url]/*</td>
</tr>
</tbody>
</table>

### TABLE V QUERIES FOR XMARK

<table>
<thead>
<tr>
<th>query</th>
<th>XPath expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q11</td>
<td>//site/open_auction[.//seller_person]/.bidder/</td>
</tr>
<tr>
<td>Q12</td>
<td>//site/open_auction[.//seller_person]/.bidder_increase/</td>
</tr>
<tr>
<td>Q13</td>
<td>//site/open_auction[.//seller_person]/.bidder_increase/.//initial]*/description/</td>
</tr>
<tr>
<td>Q14</td>
<td>//site/open_auction[.//seller_person]/.bidder_increase/description/</td>
</tr>
<tr>
<td>Q15</td>
<td>//site/open_auction[.//seller_person]/.bidder_increase/description/*</td>
</tr>
</tbody>
</table>

To avoid the frequent use of the axes like following-sibling in the tables, we assume that the order between the siblings in a tree query follows the left-to-right order in the corresponding XPath expression. For example, //inproceedings[key][author] indicates that key is followed by author.

In this test, we measure the CPU time performance of the streaming algorithms as $t = t_{\text{total}} - t_{\text{I/O}}$, where $t_{\text{total}}$ is the total running time and $t_{\text{I/O}}$ is the time for reading (from disk into memory) XML documents and storing the results on disk.

In Fig. 11, we show the running time of all the methods for TreeBank, which shows that Chen-1 outperforms all the other methods. TwigM has the worst performance while Gou’s and Chen-2 are comparable. For Q4, Chen-2 is clearly worse than Gou’s. It is because in the presence of ‘*’, the checking of the left-to-right relationships uses extra time, but does not filter a significant amount of false drops. However, for the other four queries, Chen-2 works slightly better than Gou’s.

In Fig. 12 and 13, we show the whole execution times for processing queries against DBLP and XMark, respectively. These two figures also that Chen-1 has the best performance.
the process of a query evaluation, a huge number of potential answer nodes may have to be buffered. In Fig. 14, 15, and 16, we show the sizes of the buffering spaces used during the evaluation of queries against TreeBank, DBLP, and Xmark, respectively. From these figures we can see that the space requirement of Chen-2 is the lowest among all the tested methods for the following reasons:

1. By Chen-2, not only the ancestor/descendant, but also left-to-right relationships are used to get rid of non-qualifying nodes.
2. Before a node in \( T \) is checked, only for each of its children the corresponding matching query nodes are buffered.
3. Using the merging operation, the buffering space is effectively reduced.

In addition, we notice that the buffering space of Chen-1 is also much smaller than Gou’s due to (2) and (3) listed above.

![Fig. 14 Size of buffering space for TreeBank](image1)

![Fig. 15 Size of buffering space for DBLP](image2)

![Fig. 16 Size of buffering space for XMark](image3)

VI. CONCLUSIONS

In this paper, two efficient algorithm for the query evaluation in an XML streaming environment is presented. One is for the unordered tree matching. The algorithm runs in \( O(|T| \cdot \text{leaf}_T) \) time and \( O(|T| \cdot \text{leaf}_Q) \) space, where \( \text{leaf}_T \) stands for the number of the leaf nodes in \( T \) and \( \text{leaf}_Q \) for the number of the leaf nodes in a query tree \( Q \). The other is for the ordered tree matching, by which the left-to-right order of nodes much also be taken into account. It runs in time and space complexities are bounded by \( O(|T| \cdot |Q|) \). But its space overhead is in the order of \( O(\text{leaf}_T \cdot |Q|) \). These computational complexities are much better than any existing strategy for this problem.

REFERENCES


Yangjun Chen received his BS in Information System Engineering from the Technical Institute of Changsha, China, in 1982, and his Diploma and PhD in Computer Science from the University of Kaiserslautern, Germany, in 1990 and 1995, respectively. From 1995 to 1997, he worked as a Post-Doctor at the Technical University of Chemnitz-Zwickau, Germany. After that, he worked as a Senior Engineer at the German National Research Center of Information Technology (GMD) for more than two years. Since 2000, he has been a Professor in the Department of Applied Computer Science at the University of Winnipeg, Canada. His research interests include deductive databases, federated databases, document databases, constraint satisfaction problem, graph theory and combinatorics. He has more than 150 publications in these areas.

Leping Zou received his BS from the South-West JiaoTong University of China, in 2003. He is a graduate student in the Department of Applied Computer Science, University of Winnipeg, Canada.