A Rapid Tree Pattern Matching Algorithm for XML Query

P.Radha Krishna M.Tech
ASSOCIATE Professor,
Dept of Computer Science and Engg.,
NOVA College of Engineering &Technology
Ibrahimpatnam, Vijayawada

P.Aslesha
Final M.Tech Student
Dept of Computer Science and Engg.,
NOVA College of Engineering &Technology
Ibrahimpatnam, Vijayawada
Email: slesha.ponnaganti@yahoo.com

S.Rajitha
Final M.Tech Student
Dept of Computer Science and Engg.,
NOVA College of Engineering &Technology
Ibrahimpatnam, Vijayawada

ABSTRACT
Finding all distinct matching’s of the query tree pattern is the main operation of XML query evaluation. The existing methods 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 for tree pattern matching are decomposition matching-merging processes, which may produce large useless intermediate result or require repeated matching of some sub-patterns. We propose a fast tree pattern matching algorithm called TreeMatch to directly find all distinct matchings of a query tree pattern. The only requirement for the data source is that the matching elements of the non-leaf pattern nodes do not contain sub-elements with the same tag. The TreeMatch does not produce any intermediate results and the final results are compactly encoded in stacks, from which the explicit representation can be produced efficiently. It can effectively control the size of intermediate results during query processing.

Keywords— leaf pattern, Query processing, XML, algorithms, tree pattern , Query patterns.

I. INTRODUCTION
The XML (eXtensible Markup Language) is gaining popularity as a new standard for data representation and exchange on the internet. The problem of querying XML documents has been given much attention by researchers. The XML documents can be considered as semi-structured databases. There are generally two streams of query approaches to the XML data. The first one is to map the XML documents into relational databases in order to use a structured query languages such as SQL. It usually produces too many relations and loses some important information on relationships. For example, the explicit hierarchical relationship between XML elements may be lost. The second one is to design new query languages to extract information from XML documents. These query languages query not only the contents but also the structure. The XML documents are usually modelled as

trees and queries in XML query languages and are typically twig (or small tree) patterns with some nodes having value-based predicates. Therefore, finding all distinct matchings of a twig pattern becomes a core operation in an XML query evaluation.

The existing methods for tree pattern matching in XML are typically a decomposition-matching-merging process. The drawback of the decomposition-matching-merging methods is that the size of intermediate results may be much larger than the final answers. The main reason of having larger intermediate results and repeated matching of sub-patterns is due to the consideration of self-containment XML documents, i.e., an XML element that has the same tag with its sub-elements. However, in the real applications, we seldom find self-containment documents. We propose a fast tree matching algorithm called TreeMatch that can directly find all matchings of a tree pattern in one step. The only requirement for the data source is that the matching elements of the non-leaf pattern nodes do not contain sub-elements with the same tag.

II. DEFINITIONS
The paper is organized as follows:

Definition1: Introduces the fundamentals of XML query and related work for XML query evaluation.

Definition2: Discusses in detail the fast tree pattern match algorithms TreeMatch.

Definition3: is the XML Document

Therefore, it does not produce any intermediate results and does not need the merging process. Second, the final results are compactly encoded in stacks and explicit representation of the results, either as a tree or a relation with each tuple representing one matching, can be generated efficiently.

Def1: XML Query’s:
Many company and enterprises produce and swap over over XML data more often, there is an increasing need for efficient processing of queries on XML data. XPath query: A[B]/C and the corresponding XML tree pattern. This
query finds all node C that has the parent A which has another child B. In Figure 1(b), the query answers are nodes "C₁" and "C₂".

It denotes the return node in query. The answers are C₁ and C₂. The digital labels will be explained later. Efficient matching of XML tree patterns has been usually considered as a core operation in XML query processing. In recent years, many methods ([13], [14], [25]) have been proposed to match XML tree queries efficiently.

Fig. 1. Example XML tree query and document.

**Def 2:** Tree Pattern

Tree Pattern Match finds all the matchings of the pattern by recursively calling function find(q) to find the matchings of subpattern rooted by q and compactly encodes the matchings in the stacks associated with each pattern node. At any time the stacks associated with pattern node q and its descendants contain the matching nodes of sub-pattern rooted by q that are possibly extended to the final result.

![Pattern Tree](image)

**Def 3:** XML Document

The XML (eXtensible Markup Language) is gaining popularity as a new standard for data representation and exchange on the internet. The problem of querying XML documents has been given much attention by researchers.

**III. HOLISTIC ALGORITHMS**

In this section, In Existing system an algorithm to evaluate an extended XML tree query. In the algorithm is to achieve a large optimal query class according to afore mentioned theorems.

3.1 TreeMatch for Q^/∗^/∗^/∗^/∗^/∗^.

3.1.1 Data structures and notations

There is an input list Tq associated with each query node q, in which all the elements have the same tag name q. Thus, we use eq to refer to these elements. cur(Tq) denotes the current element pointed by the cursor of Tq. The cursor can be advanced to the next element in Tq with the procedure advance(Tq). There is a set Sq associated with each branching query node q (not each query node). Each element eq in sets consists of a 3-tuple (label, bit Vector, output List). label is the extended Dewey label of eq. bitVector is used to demonstrate whether the current element has the proper children or descendant elements in the document. Specifically, the length of bitVector(eq) equals to the number of child nodes of q. Given a node qc ∈ children(q), we use bitVector(eq)[qc] to denote the bit for qc. Specifically, bitVector(eq)[qc]="1" if and only if there is an element eqc in the document such that the eq and eqc satisfy the query relationship between q and qc.

In our algorithm, we will frequently use the following two notations. (1) NAB(q) denotes the Nearest Ancestor Branching node of q in the query pattern Q. Formally, q=NAB(q) if and only if q′ is a branching node and q′ is an ancestor of q and there is no other branching node q′′ such that q′′ is in the path from q′ to q.

![Set Encoding](image)

**Algorithm 1:** Algorithm TreeMatch for class Q^/∗^/∗^/∗^/∗^/∗^.

1: locateMatchLabel(Q);
2: while (~end(root)) do
3: fact = getNext(topBranchingNode);
4: if (fact is a return node) addToOutputList(NAB(fact), cur(Tfact));
5: updateSet(fact);
6: locateMatchLabel(Q);
7: emptyAllSets(root);
IV. THE XML QUERY FUNDAMENTALS

We propose the background information of the XML query and notations used in this paper. An XML document consists of nested elements enclosed by user-defined tags, which indicate the meaning of the content contained. Figure 1 shows an example of an XML document named “pub.xml”, which contains some publication information. The hierarchical structure of an XML document can be modeled as a tree. Figure 3 is the tree representation of the XML file in Figure 1. The XML documents on the Internet is an forest of XML trees and we call it an XML database.

```xml
<?xml version="1.0" ?>
<publication>
  <journal title="DBMS">
    <article>
      <title>Index Construction</title>
      <author>Smith</author>
    </article>
  </journal>
  <journal title="Algorithm">
    <article>
      <title>DBMS</title>
      <editor>Jack</editor>
    </article>
  </journal>
</publication>
```

Figure 1. An example of an XML document

The semi-structured format of XML documents brings the possibility of using database technology to query the XML data in its place of information recovery techniques applicable only to plain text documents. However, the mature SQL queries cannot be applied directly since XML documents do not necessarily conform to a predefined, rigid schema required by the traditional database system. Much research has been done on XML query languages. Although the query languages differ in detailed grammars, they have a common feature, that is: querying structure as well as the contents or values of elements. Queries in XML query languages make use of tree patterns to match portions of data in the XML database. For example, the following is a query expressed in Xquery over the document in Figure 1 where “/” indicates ancestor-descendant relationship, and “/” indicates parent-child relationship.

```
FOR $a IN document("http://.../pub.xml")//journal/article
    $b IN $a/title
    WHERE $a/author = "Smith" RETURN
    <article>$b</article>
```

This query retrieves the titles of articles authored by “Smith” and published in a journal. It contains both structure and content information. We can use a tree to depict the query as shown in Figure 2. In other words, this query will find all the matching’s of the tree pattern in the XML.

Most research in literature focuses on the second subproblem: find all matchings of a linear pattern. It can be classified according to the type of linear pattern they deal with, i.e., matching the binary structural relationships and matching path patterns. To match the binary structural relationships, Zhang et al. [16] proposed the MPMGJN (multipredicate Merge Join) algorithm and Al-Khalifa et al. [9] gave the Stack-Tree algorithms. The algorithms accept two lists of sorted individual matching nodes and structurally join pairs of nodes from both lists to produce the matchings of the binary relationships. The difference between the MPMGJN and Stack-Tree is that the MPMGJN is a variation of the traditional merge join algorithm, requiring multiple scans of the input lists. The Stack-Tree algorithm is more efficient as it uses stacks to maintain the ancestor or parent nodes and it needs only one scan of the input lists. The former is for matching path patterns and the latter is claimed to solve the problem of twig pattern matching. Both of them use a chain of stacks to encode the partial result. However, TwigStack does not match the twig pattern directly. It still belongs to the decomposition-matching-merging category. All the algorithms discussed above use the format.

```
Figure 2. The tree representation of the example query
Figure 3. The tree representation of the XML document example

The existing methods for tree pattern matching in XML is typically a decomposition-matching-merging process:
1) Decompose the tree pattern into linear patterns which might be binary (parent-child or ancestor-descendant) relationships between pairs of nodes or root-to-leaf paths;
2) Find all matchings of each linear pattern; and
3) Merge-join them to produce the result.
```
The advantage of using (DocId; Start : End; Level) to represent nodes is that we can determine the relationships between nodes in a constant time. A simple definition of the terms ancestor, descendant, parent and child is given as follows:

**Definition 1** Suppose that x and y are two nodes from an XML tree, we say that y is an ancestor of x and x is a descendant of y if y.DocId = x.DocId and y.Start < x.Start < y.End. y is a parent of x and x is a child of y if 1) y.DocId = x.DocId. 2) y.Start < x.Start < y.End, and 3) y.Level = x.Level - 1.

For example, in Figure 3, the author node (1; 13 : 15; 3) is a descendant of the journal node (1; 2 : 17; 1).

The TwigStack algorithm partially solved the problem of larger intermediate results with decomposition matching-merging methods. When the patterns have only ancestor-descendant edges, the intermediate result of each path matching is guaranteed to be part of the final result. However, TwigStack’s requirement of matching all the root-to-leaf paths leads to repeated matching of the common nodes shared by multiple paths. If the query twig pattern has \(N\) leaf nodes, there will be \(N\) different root-to-leaf paths. The difficulty of directly matching tree patterns comes from the self-containment property of the XML elements, that is, elements have the same tag with their sub elements. When the patterns have only ancestor-descendant edges, the intermediate result of each path matching is guaranteed to be part of the final result. However, TwigStack’s requirement of matching all the root-to-leaf paths leads to repeated matching of the common nodes shared by multiple paths. If the query twig pattern has \(N\) leaf nodes, there will be \(N\) different root-to-leaf paths. However, self-containment is seldom found in real. Evaluating the query expression reduces to finding distinct matches of the GTP.

V. THE TREE MATCH ALGORITHM

5.1 Problem Definition

We first introduce some basic notations before giving the formal definition of the problem. An XML document is modelled as a tree \(D = (Nd; Ed)\), where \(Nd\) is a set of 3 labelled nodes, \(Ed\) is a set of edges. We call \(D\) a data source. For a node \(x \in Nd\), the label of \(x\) is denoted by \(labelD(x)\).

![Figure 4. The data source](image)

A query tree pattern is a tree \(Q = (Nq; Eq)\), where \(Nq\) is a set of labelled nodes and \(Eq\) is a set of edges. Each edge is represented by the pair of nodes it connects. There are two kinds of edges in \(Eq\), parent-child edges and ancestor-descendant edges. An ancestor-descendant edge is represented by \(x == y\) and a parent-child edge is represented by \(x = y\), where \(x; y \in Nq\) are the nodes connected by the edge. For a node \(x \in Nq\), \(labelQ(x)\) denotes the label of \(x\).

**Definition 2** Given two nodes \(x \in Nq\) and \(x' \in Nq\), we say that \(x'\) is an occurrence of \(x\) in \(D\) if \(labelQ(x) = labelD(x')\).

**Definition 3** Given a query tree pattern \(Q = (Nq, Eq)\) and a data source \(D = (Nd, Ed)\), for any node \(q \in Nq\), if there exist two occurrences of \(q\) that have ancestor-descendant or parent-child relationship, \(q\) is said to have occurrence with self-containment.

**Definition 4** Given a query tree pattern \(Q = (Nq, Eq)\) and a data source \(D = (Nd, Ed)\), we call \(\psi(Nq)\) a matching of \(Q\) in \(D\) if there is a mapping \(\psi : Nq \rightarrow Nd\) that satisfies the following properties for every \(x, y \in Nq\):

1. \(x \neq y \iff \psi(x) \neq \psi(y)\);
2. \(labelQ(x) = labelD(\psi(x))\);
3. if \(x/y \in Eq \implies \psi(x)\) is an ancestor of \(\psi(y)\); if \(x/y \in Eq \implies \psi(x)\) is the parent of \(\psi(y)\);

The TreeMatch algorithm works in the condition of no occurrence of self-containment. In other words, the problem is defined as: given a tree pattern \(Q\) where each non leaf node does not have occurrence with self-containment, and a XML data source \(D\) that has index structures to identify occurrences of \(Q\)’s nodes, find all the distinct matchings of \(Q\) in \(D\).

5.2 Algorithm

All the nodes of \(Tq\) are occurrences of the pattern node \(q\) in data source. Function \(Find(q)\) is called to determine whether the current occurrence \(Tq/current\) is a partial solution. \(Tq/current\) is a partial solution means matchings of sub-tree patterns rooted by \(q\) have been found and encoded in the stacks and these matchings are possibly extended to final results. If \(Tq/current\) is found not to be a partial solution, function \(CleanStack()\) is called to remove the recoded nodes that are descendants of \(Tq/current\). Function \(GenerateSolution()\) and \(GenerateSolution2()\) produce two varieties of explicit representation of the final result.

**TreeMatch(q)**

While (\(Tq\) is not empty)

- if (\(Find(q)\)) push(\(Sq, Tq/current, root\));
- \(Advance(Tq)\);
- \(GenerateSolution(q)\);
- \(Find(q)\)
Partial solutions can be recursively defined as follows:

**Definition 5:** The occurrences of leaf pattern nodes are partial solutions. An occurrence $x_i$ of a non-leaf pattern node $x$ is a partial solution if 1) $x_i$ has child nodes that are the occurrences of each child node of $x$; and 2) these child nodes of $x_i$ are partial solutions.

- **Algorithm:****

  ```plaintext
  def CleanStack(q):
      N = NumOfChildren(q);
      i = 0;
      while (i++ < N) CleanStack(qi);
      ParentNode = Sqi.top parent;
      While(Sqi is not empty AND Sqi.top parent == ParentNode)
      Pop(Sqi);

  def GenerateSolution(q):
      N = NumOfChildren(q);
      i = 0;
      while (i++ < N) GenerateSolution(qi);

  def GenerateSolution2(q, parent)
      while(Sqi.bottom parent == parent)
      AppendToSolution(Sqi.bottom parent);
      N = NumOfChildren(q);
      i = 0;
      while (i++ < N) GenerateSolution2(qi, Sqi.bottom parent)
      Sqi.bottom parent = Sqi.bottom parent;
      AppendToSolution(-1);
      AppendToSolution(-1);
  ```

For example, given a tree pattern as shown in Figure 5 and data source in Figure 4, $b_1$, $b_2$, $b_3$, $b_4$ are all partial solutions since $b$ is a leaf pattern node. $c_2$ is also a partial solution since its children $d_2$, $e_1$ match the children of node $c$, and $d_2$, $e_1$ are partial solution. In comparison, $c_1$ is not a partial solution and $a_1$ is not a partial solution. According to above definition, a matching node is a partial solution if its matching child nodes are partial solutions. (Start; End; Level) representation of node is used to determine the relationship between nodes in $T_q$ and $T_{qi}$. The algorithm checks only the ancestor/descendant relationships. To deal with the parent/child relationship, we only need to test the Level of nodes by replacing line (15) with “if ($T_{qi}.current.Level = T_q.current.Level + 1$ AND Find(qi))”. If $T_{qi}.current.start$ is greater than $T_q.current.end$ (line 5), node $T_{qi}.current$ and its followers in $T_{qi}$ are not descendants of $T_q.current$. At this point, the algorithm has to check the present status before taking further action. If there is a node before $T_{qi}.current$ that is a partial solution, the algorithm starts to check $T_{qi+1}$ (lines 6-8). Otherwise, the algorithm cleans the stack $Sqi_1$, $Sqi_2$, ..., $Sqi$ by calling function CleanStack since the nodes in the stacks that are descendants of $T_q.current$ cannot be extended to produce a final result.

The `GenerateSolution()` function is called to produce explicit representation of the final result from the encoded one. It generates the distinct matchings one by one. The `Join(Sq, Sqi)(line 5)` is an operation equivalent to the equal join in relational databases, the content of $Sq$ and $Sq_i$ being regarded as relations. The `GenerateSolution2()` function gives a concise representation of the final result. It organizes the matchings as a tree, and produces the string representation of the tree as illustrated in Figure 8. The string representation can uniquely determine the tree.
VI. CONCLUSIONS

The paper proposes a TreeMatch algorithm to directly find all distinct matchings of a query tree pattern in XML data sources. Unlike prior research for query tree pattern matching, the TreeMatch algorithm does not need to decompose the tree pattern into linear patterns and does not produce any intermediate results that are not part of the final results. The TreeMatch algorithm is applicable when the non-leaf pattern nodes do not have occurrences with self-containment. The self-containment is seldom found in real XML documents and such a property can easily be identified. Therefore, the TreeMatch algorithm is more efficient than the existing methods under most cases.

REFERENCES


AUTHORS PROFILE

P.ASLESHA has completed B.Tech from NOVA College of Engineering Technology for Women in 2010 and Pursuing M.Tech in NOVA College of Engineering Technology for Women. Her research work includes concepts of cloud computing used for Linear and Non-Linear Programming in Cloud Computing.

Mr. P. Radha Krishna did his Master in Computer Application at Lakkireddy Bali eddy college of engineering. He started his teaching career at Anurag Engineering College, Kodad. Where He had guided be so many batches in their projects. He guided students in doing IBM projects at Anurag. He played an active role in the NBA Accreditation process at Anurag Engineering College in to 2007. He had acted as In charge HOD of MCA at Anurag. Mr. P Radha Krishna is a Members of ISTE .He had participated many workshops and paper presentations, which include Sunmicrosystem and open systems (LAMP) workshops .Mr. Radha Krishna is the secretary of Alumni Association of LBRCE .He is now working as the Head of the Department of CSE of Nova college of Engineering & Technology, Jupudi, Ibrahimpatnam.

S. RAJITHA has completed B.Tech from NOVA College of Engineering Technology for Women in 2010 and Pursuing M.Tech in NOVA College of Engineering Technology for Women. Her research work includes concepts of Data Mining and Data Warehousing used for Collective Behavior of social networking information by using Edge Clustering.