Evaluating Partial Tree-Pattern Queries on XML Streams

Xiaoying Wu
New Jersey Institute of Technology, USA
xw43@njit.edu

Dimitri Theodoratos
New Jersey Institute of Technology, USA
dth@cs.njit.edu

ABSTRACT

The streaming evaluation is a popular way of evaluating queries on XML documents. Besides its many advantages, it is also the only option for a number of important XML applications. Unfortunately, existing algorithms focus almost exclusively on tree-pattern queries (TPQs). Requirements for flexible querying of XML data have motivated recently the introduction of query languages that are more general and flexible than TPQs.

We consider a partial tree-pattern query (PTPQ) language which generalizes and strictly contains TPQs. PTPQs can express a fragment of XPath which comprises reverse axes and the node identity equality (\(=\)) operator, in addition to forward axes, wildcards and predicates. We outline an original streaming algorithm for PTPQs. Our algorithm is the first one to support the streaming evaluation of such a broad fragment of XPath.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems—query processing, textual databases

General Terms: Algorithms.

Keywords: XPath query evaluation, XML

1. INTRODUCTION

Many recent research efforts have focused on developing evaluation algorithms for different subclasses of XPath. A large number of these algorithms consider evaluating queries over streaming XML documents [2, 3]. Unfortunately, existing algorithms on XML streams focus almost exclusively on tree-pattern queries (TPQs). A distinguishing restrictive characteristic of TPQs is that they impose a total order for the nodes in every path of the query pattern.

We consider a query language for XML, called partial tree-pattern query (PTPQ) language. The PTPQ language generalizes and strictly contains TPQs. PTPQs are not restricted by a total order for the nodes in a path of the query pattern since they can constrain a number of (possibly unrelated) nodes to lie on the same path (same-path constraint). They are flexible enough to allow on the one side keyword-style queries with no structure, and on the other side fully specified TPQs.

Figure 1: A PTPQ and its graph representation

In this paper, we undertake the task of designing an efficient evaluation streaming algorithm for PTPQs. This task is complex: as we show later, due to their expressive power, PTPQs can only be represented as directed acyclic graphs (dags) annotated with same-path constraints. Matching these query patterns to XML trees requires the appropriate handling of the structural constraints of the dag, and the same-path constraints.

2. QUERY LANGUAGE

Syntax and Semantics. A partial tree-pattern query (PTPQ) specifies a pattern which partially determines a tree. PTPQs comprise nodes and child and descendant relationships among them. Their nodes are grouped into disjoint sets called partial paths. PTPQs are embedded to XML trees. The nodes of a partial path are embedded to nodes on the same XML tree path. However, unlike paths in TPQs the child and descendant relationships in partial paths do not necessarily form a total order. This is the reason for qualifying these paths as partial. PTPQs also comprise node sharing expressions. A node sharing expression indicates that two nodes from different partial paths are to be embedded to the same XML tree node. That is, the image of these two nodes is the same – shared – node in the XML tree. Figure 1(a) shows a PTPQ.

The answer of a query on an XML tree is a set of results, where each result is the image of the output node in a match of the query on the XML tree. The formal definitions of a PTPQ and its embedding in an XML tree can be found in the full version of the paper [1].

Graph representation for PTPQs. We represent queries as node and edge labeled directed graphs: a query \(Q\) is represented by a graph \(Q_G\). Every node \(X\) in \(Q\) corresponds to a node \(X_G\) in \(Q_G\). Node \(X_G\) is labeled by the label of \(X\). Two nodes in \(Q\) participating in a node sharing expression correspond to the same node in \(Q_G\). Otherwise, they correspond to distinct nodes in \(Q_G\). A node in \(Q_G\) is labeled by "*" if all the nodes in \(Q\) it corresponds to are labeled by "*". In addition, each node in \(Q_G\) is annotated by the set of PP nodes in \(Q\) it corresponds to. These annotations express same path constraints. For every structural relationship \(X/Y\)
not contribute to a possible new matching for the output node are ignored and not pushed on stacks. For instance, consider evaluating the query $Q_2$ of Figure 2(b) on the XML tree of Figure 2(a).

The nodes $a_1, b_1, e_1$ and $f_1$ which are matches for the predicate nodes $A, B, E$ and $F$, respectively, contribute to the match $d_1$ of the output node $D$. The nodes $a_2, \ldots, a_n, b_2, \ldots, b_n, e_2, \ldots, e_n, f_2, \ldots, f_n$ which are also matches of the predicate nodes can be ignored, since they all contribute to the same match $d_1$ of the output node. Note that these nodes correspond to $O(n^4)$ embeddings of the query with the same match for the output node. Avoiding their computation saves substantial time and space.

![XML Tree](a), (b) Query $Q_2$

To deal with the same path constraint, procedure `startEval` checks if the stacks contain an entry for every `sink` node $X$ of those partial paths $p_i$ (i.e., $X$ is annotated by $p_i$ but none of its descendants in $Q$ does) and update some data structures to record such information.

**Close event handler.** When the algorithm receives the close event of $x$, it calls the close event handler procedure `endEval` on the sequence of compatible query nodes (line 9). For each query node $X$ in the list, procedure `endEval` pops out the entry of $x$ from $S_X$ and checks if $x$ is a candidate match of $X$. Node $x$ is called a candidate match of $X$ if $x$ is the image of $X$ under an embedding of $Q_X$ to the subtree rooted at $x$ in $T$ (i.e., $x$ satisfies both the structural and the same path constraints of query node $X$). If this is the case and $X$ is a backbone node, each candidate output (a candidate match of the output node) stored in the entry of $x$ is propagated to an ancestor of $x$ in a stack, if $X$ is not the root of $Q$, or is returned to the user, otherwise. It is possible that $X$ has more than one parent node in $Q$, the stack for each of the parent nodes contains entries that are ancestors of $x$ in $T$. A key issue here is how to avoid duplicate outputs. Our strategy is to propagate the candidate outputs from $x$ only to the lowest ancestor of $x$ in the parent stacks of $X$. If $x$ is not a candidate match of $X$, the list of candidate output stored in the entry of $x$ is either propagated to an ancestor of $x$ in a stack, or is discarded, depending on whether there exists a path in stacks which consists of nodes that are candidate matches of the query nodes.

One important feature of Algorithm `PSX` is that it keeps only one copy of each candidate output in the stacks during execution. Another important feature which is especially useful in streaming environments is that the results are outputted as soon as they are available.

3. **EVALUATION ALGORITHM**

In this section, we outline our streaming evaluation algorithm for PTPQs. Let $Q$ be the input query to be evaluated on a stream of SAX [4] events for an XML tree $T$. We assume that a topological order (i.e., a linear order of the query nodes which respects the partial order induced by the structural relationships of the query) for the nodes of $Q$ is fixed with the root node $R$ of $Q$ being the first node. The algorithm is called Partial TPQ Streaming evaluation on XML (PSX) and is shown in Listing 1. Algorithm `PSX` is event-driven: as SAX events arrive, event handlers (which can be the procedures `startEval` or `endEval`) are called on a sequence of query nodes that are compatible with the current node. A query node is compatible with the current node if its label is ‘*’ or if its label is the same as that of the current node. Algorithm `PSX` is stack-based. With every query node $X$ in $Q$, it associates a stack $S_X$.

**Open event handler.** When the algorithm receives an open event of a tree node $x$, it calls the open event handler procedure `startEval` on all the query nodes in $Q$ that are compatible with $x$ (line 6). For each such node $X$, procedure `startEval` examines whether $x$ can be pushed on stack $S_X$ and whether the path of $x$ in $T$ satisfies the same path constraint for those partial paths annotating $X$. Because the answer of a query comprises only the embeddings of the output node of the query, we might not need to identify all the matches of the query pattern when computing the answer of the query. In this sense, we take advantage of the existential semantics of the query during evaluation: whenever a matching of a predicate node in the query is found, other matches of the same node that do

<table>
<thead>
<tr>
<th>Listing 1 Algorithm PSX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 while (event stream generates more events e) do</td>
</tr>
<tr>
<td>2 let x denote the tree node corresponding to e</td>
</tr>
<tr>
<td>3 compute the list <code>nodesList</code> of query nodes in Q whose labels are either ‘* ‘or the same as the label of x and are sorted by their topological order in Q</td>
</tr>
<tr>
<td>4 if (e is an open event) then</td>
</tr>
<tr>
<td>5 for (every $X \in nodesList$ in the reverse topological order) do</td>
</tr>
<tr>
<td>6 <code>startEval(X, x)</code></td>
</tr>
<tr>
<td>7 else if (e is a close event) then</td>
</tr>
<tr>
<td>8 for (every $X \in nodesList$ in the forward topological order) do</td>
</tr>
<tr>
<td>9 <code>endEval(X, x)</code></td>
</tr>
</tbody>
</table>

(resp. $X//Y$) in $Q$ there is a single (resp. double) edge in $Q_C$. Figure 1(b) shows the query graph of query $Q_1$ of Figure 1(a). For simplicity of presentation, the annotations of some nodes might be omitted and it is assumed that a node inherits all the annotating PPs of its descendant nodes. For example, in the graph of Figure 1(b), node $A$ is assumed to be annotated by the PPs $p_1, p_2$, and $p_3$ inherited from its descendant nodes $D, E$, and ‘*’.

**Properties of Partial-Tree Pattern Query Language.** Clearly, the class of PTPQs cannot be expressed by TPQs. TPQs correspond to the fragment XP[//]/(i.e., * of XPath that involves predicates(i), child (i) and descendant (i) axes, and wildcards (*). PTPQs represent a very broad fragment XP[[//]/~\(~\)](i.e., XP[//]/\(\times\)~\(~\)] of XPath that corresponds to XP[[//]/~\(~\)] augmented with the is operation (\(\times\)) of XPath. The is operator is a node identity equality operator.

As we show in [1], a PTPQ is equivalent to a set of TPQs. One could then consider evaluating PTPQs using existing streaming algorithms for TPQs. The number of TPQs that need to be evaluated could then consider evaluating PTPQs using existing streaming algorithms for TPQs. The number of TPQs that need to be evaluated can be exponential on the number of nodes of the PTPQ. Clearly, it is inefficient to evaluate a PTPQ by evaluating an exponential number of TPQs. Therefore, previous streaming algorithms cannot be used for efficiently evaluating PTPQs.

4. **REFERENCES**


