Web Data Management

Tree Pattern Evaluation

Serge Abiteboul
INRIA Saclay & ENS Cachan

Ioana Manolescu
INRIA Saclay & Paris-Sud University

Philippe Rigaux
CNAM Paris & INRIA Saclay

Marie-Christine Rousset
Grenoble University

Pierre Senellart
Télécom ParisTech

Contents

In this chapter, we learn how to build an evaluation engine for tree-pattern queries, using the SAX (*S*imple *A*PI for *X*ML) programming model. We thereby follow a dual goal: (i) improve our understanding of XML query languages and (ii) become familiar with SAX, a stream parser for XML, with an event-driven API. Recall that the main features of SAX were presented in Section **??**.

1 Tree-pattern dialects

We will consider tree-pattern languages of increasing complexity. We introduce them in this section.

C-TP This is the dialect of *conjunctive tree-patterns*. A *C-TP* is a tree, in which each node is labeled either with an XML element name, or with an XML attribute name. C-TP nodes corresponding to attributes are distinguished by prefixing them with `@`, e.g., `@color`. Each node has zero or more children, connected by edges that are labeled either / (with the semantics of child) or // (with the semantics of descendant). Finally, the nodes that one wants to be *returned* are marked.

As an example, Figure 1 shows a simple XML document d where each node is annotated with its preorder number. (Recall the definition of this numbering from Section **??**.) Figure 2 shows a C-TP pattern denoted t_1 and the three tuples resulting from "matchings" of t_1 into d. A *matching* v is a mapping from the nodes in the tree-pattern to the nodes in the XML tree that verifies the following conditions: For each nodes n, m,

- If n is labelled l for some l, $v(n)$ is an element node labelled l; If n is labelled $@l$, $v(n)$ is an attribute node labelled $@l$;
- If there is a / edge from n to m, $v(n)$ is a parent of $v(m)$; If there is a // edge from n to m, $v(n)$ is an ancestor of $v(m)$.

In the figure, the nodes that we want to be returned are marked by boxes surrounding their labels. Observe that a result is a tuple of nodes denoted using their preorder numbers. For now, assume that C-TPs return tuples of preorder numbers. In real-world scenarios, of course, we may also want to retrieve the corresponding XML subtrees, and at the end of this chapter, the reader will be well-equipped to write the code that actually does it.

Before moving on and extending our language of tree-pattern queries, we next observe an important aspect of the language. For that consider the following three queries:

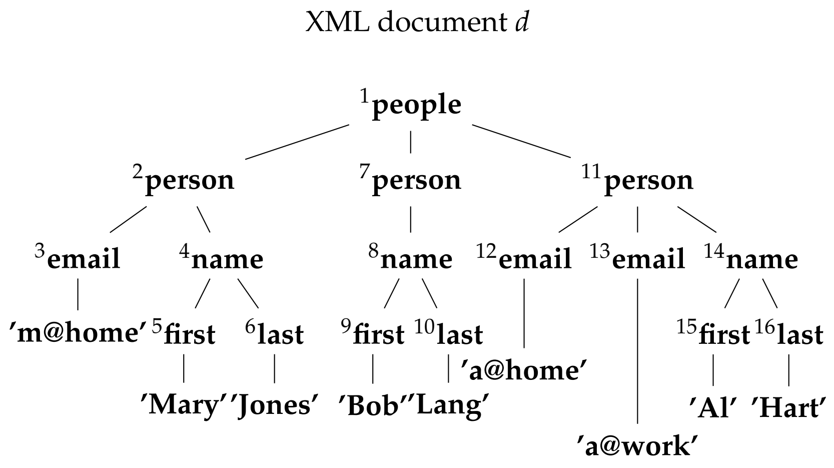


Figure 1: A sample document.

Query q_1:
```
for $p in //person[email]
                  [name/last]
return ($p//email,
        $p/name/last)
```

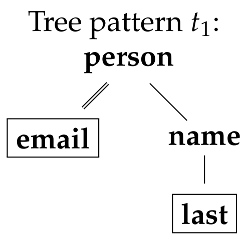
Query q_1':
```
for $p in //person
return ($p//email, $p/name/last)
```

Query q_2:
```
for $p in //person[name/last]
return ($p//email,
        $p/name/last)
```

Which one do you think corresponds to the tree-pattern query t_1? Well, it is the first one. In q_1, the `for` clause requires the elements matching `$p` to have both an e-mail descendant, and a descendant on the path `name/last`. Similarly, to obtain a matching from t_1 and d, we need matches on both paths. In contrast, consider the more relaxed query q_1' that would output the last name of a `person` element without an `email`, or the `email` of a `person` element without last name. Lastly, query q_2 requires a last name but no email. This motivates an extension of the language we consider next.

TP The dialect TP is a superset of C-TP, extending it to allow optional edges. Syntactically, TP distinguishes between two kinds of edges, compulsory and optional edges. In a nutshell, the semantics of TP is defined as follows. Matchings are defined as for C-TP. The only difference is that for an optional child edge from n to m, two cases can be considered:

- If $v(n)$ has some child $v(m)$ such that there is a matching from the subtree of the query rooted at m and the subtree of d rooted at $v(m)$; then v extends such a matching from



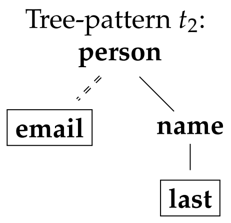
Embeddings of t_1 in d

person	email	name	last
2	3	4	6
11	12	14	16
11	13	14	16

Expected result $t_1(d)$

email	last
3	6
12	16
13	16

Figure 2: A C-TP and its result for a sample document.



Embeddings of t_2 in d

person	email	name	last
2	3	4	6
7	null	8	10
11	12	14	16
11	13	14	16

Expected result $t_2(d)$

email	last
3	6
null	10
12	16
13	16

Figure 3: A TP and its result for the sample document in Figure 1.

the m-subtree to the $v(m)$-subtree.

- Or $v(n)$ has no such child, then v has a null value for m and all its descendants.

And similarly for descendant.

As an example, Figure 3 shows the TP pattern t_2 corresponding to the previously see query q_2. It resembles t_1 but the edge between the `person` and `email` nodes is optional (denoted by the dashed lines in the figure). As the figure shows, the second `person` element from the document in Figure 1 lacks an email, which leads to a matching tuple with a null. As a consequence, one of the tuples in $t_2(d)$ contains a null email.

To conclude this section, we consider three somehow orthogonal extensions of both TP and C-TP. Figure 4 shows the extended tree-pattern t_3, t_4 and t_5, together with their equivalent queries, respectively q_3, q_4 and q_5, and their results for the sample document in Figure 1. The three extensions are:

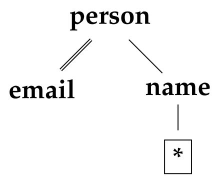
Star (*) patterns We allow labels to be not only element or attribute names, but also "*" that are interpreted as wild cards matched by any element or attribute.

Value predicate patterns We can impose a constraint on nodes using some predicates. For instance, we can impose the value of an email, e.g, `email = m@home`.

Query q_3:

```
for $p in //person[email]
return $p//name/*
```

Equivalent tree pattern t_3:



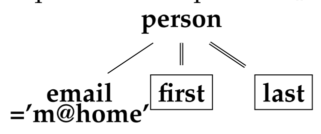
Evaluation result of $t_3(d)$:

*
5
6
9
10
15
16

Query q_4:

```
for $p in //person[//first]
                  [//last]
where $p/email='m@home'
return ($p//first,
        $p//last)
```

Equivalent tree pattern t_4:



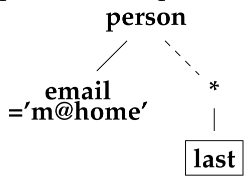
Evaluation result of $t_4(d)$:

first	last
5	6

Query q_5:

```
for $p in //person
where $p/email='m@home'
return <res>{$p/*/last}</res>
```

Equivalent tree pattern t_5:



Evaluation result of $t_5(d)$:

last
<last>Jones</last>

Figure 4: Extended patterns.

Patterns returning full elements Finally, we can request the tree-pattern to not only return the preorder values of nodes but the entire corresponding subtrees. For simplicity, we do not introduce any explicit graphical notation for requesting that in answers. In Figure 4, the result of $t_5(d)$ assumes that "full answers" are requested.

From a practical perspective of XML querying, all these extensions are interesting. In particular, C-TP with all the extensions corresponds to a large and useful subset of XPath, whereas its counterpart building on TP is at the core of an important expressive fragment of XQuery. We first consider the evaluation of C-TP. Then look at extensions.

2 CTP evaluation

We now describe an algorithm, named **StackEval**, for evaluating a C-TP pattern t on a document d in a single SAX-based traversal of the document. To understand the algorithm, it is useful to have in mind that a node matches both because it satisfies some "ancestor condition" (the path from the root has a certain pattern) and also because its descendants satisfy some "descendant conditions". We know whether an XML node satisfies some ancestor conditions, by the time we encounter the opening tag of the node, because by that time, all its ancestors have been encountered already. However, we can only decide if the node satisfies descendant conditions after the complete traversal of all its descendants, that is, when encountering the closing tag of the node.

We start by describing some data structures we will use:

- When a node n_d in document d is found to satisfy the *ancestor conditions* related to a node n_t in t, a `Match` object is created to record this information. A `Match` holds the following:

```
class Match {
   int start;
   int state;
   Match parent;
   Map <PatternNode, Array<Match>> children;
   TPEStack st;

   int getStatus() {...}
}
```

```
class TPEStack {
   PatternNode p;
   Stack <Match> matches;
   TPEStack spar;

   Array <TPEStack> getDescendantStacks(); {...}
   // gets the stacks for all descendants of p
   push(Match m){ matches.push(m); }
   Match top(){ return matches.top(); }
   Match pop(){ return matches.pop(); }
}
```

Table 1: Outline of the `Match` and `TPEStack` classes.

- the `start` number of node n_d (an integer).
- an internal `state` flag, whose value can be either `open` or `closed`.
- (ancestor conditions) the `Match parent` that corresponds to a match between the parent (or an ancestor) of n_d, and the parent of n_t in the tree pattern query. If the edge above n_t is a parent edge, then `parent` can only correspond to the parent node of n_d. Otherwise, `parent` may be built from the parent or another ancestor of n_d.
- (descendant conditions) the (possibly empty) array of matches that were created out of n_d's children or descendants, for each child of n_t in the query. Such matches for n_t children are mapped in the `children` structure on the `PatternNode` children of n_t.
- a pointer `st` to a `TPEStack` (standing for tree pattern query evaluation stack). As we will see, we associate a `TPEStack` to each node in the pattern. Then `st` points to the stack associated to the pattern node n_t.

• For each node p in the tree-pattern, a `TPEStack` is created, on which the matches corresponding to this pattern node are pushed as they are created. Observe that a `TPEStack` contains a "regular" `Stack` in which `Match` objects are pushed and from which they are popped. The extra structure of a `TPEStack` serves to connect them to each other according to the structure or the query. More specifically, each `TPEStack` corresponding to a pattern node p points to:

  - the `TPEStack spar` corresponding to the parent of p, if p is not the pattern root.
  - a set `childStacks` of the `TPEStack`s corresponding to the children of p, if any.

- The `Match` and `TPEStack` structures are interconnected, i.e., each `Match` points to the (unique) `TPEStack` on which the `Match` has been pushed upon creation.

The main features of the `Match` and `TPEStack` classes are summarized in Table 1. In our discussion of the algorithms, unless otherwise specified, we use the term "stack" to refer to a `TPEStack`.

The StackEval algorithm The algorithm evaluates C-TP queries based on the SAX XML document processing model. More specifically, the query evaluation algorithm runs suitable handlers of the methods:

```
startElement   (String nameSpaceURI, String localName, String
               rawName, Attribute attributes)
endElement     (String nameSpaceURI, String localName, String
               rawName)
```

described in Section **??**. For simplicity, within `startElement` we only use the `localName` and `Attributes`, whereas from the parameters of `endElement` we only use `localName`. As a consequence, the evaluation algorithm we describe does not take into account namespaces. Extending it to include the support of namespaces does not raise interesting algorithmic issues.

The StackEval class (Table 2) contains the stack corresponding to the query root. It also stores an element counter called `currentPre`, from which `pre` number will be assigned to new `Match` objects. Finally, it stores a stack of all `pre` numbers of elements currently open but whose end has not been encountered yet.

The `startElement` handler is notified that an element with a given `localName` and a set of attributes has just started. The handler seeks to identify the stack (or stacks) associated to query nodes which the newly started element may match. To this purpose, it enumerates all the stacks created for the query (by getting all descendants of the root stack), and for each stack it checks two conditions. The first condition is that the label of the starting node matches the label of the query nodes for which the stacks were created. A second condition applies in the case of a stack `s` created for a query node `p` having a parent in the query: we push a new match on `s` if and only if there is an open match on the parent stack of `s`, namely `p.spar`. Such an open match signifies that the newly started element appears in the right context, i.e. all the required ancestors have been matched above this element. In this case, a `Match` is created with the current `pre` number (which is incremented). The `Match` is open by default when created. Finally, the `currentPre` and `preOfOpenNodes` are updated to reflect the new element.

Since tree-patterns may also require matches in XML attributes, the next lines in the `startElement` handler repeat the previously described procedure for each of the attributes whose presence is signaled in the same call to `startElement`.

The `endElement` handler (Table 3) plays a dual role with respect to the `startElement` one. Ancestor conditions for a potential query node match are enforced by `startElement` when the element starts; descendant conditions are checked by `endElement` when the element's

```
class StackEval extends DocumentHandler {
   TreePattern q;
   TPEStack rootStack; // stack for the root of q
   // pre number of the last element which has started:
   int currentPre = 0;
   // pre numbers for all elements having started but not ended yet:
   Stack <Integer> preOfOpenNodes;

   startElement(String localName, Attribute attributes){
     for(s in rootStack.getDescendantStacks()){
       if(localName == s.p.name && s.spar.top().status == open){
         Match m = new Match(currentPre, s.spar.top(), s);
         // create a match satisfying the ancestor conditions
         // of query node s.p
         s.push(m); preOfOpenNodes.push(currentPre);
       }
       currentPre ++;
     }
     for (a in attributes){
       // similarly look for query nodes possibly matched
       // by the attributes of the currently started element
       for (s in rootStack.getDescendantStacks()){
         if (a.name == s.p.name && s.par.top().status == open){
           Match ma = new Match(currentPre, s.spar.top(), s);
           s.push(ma);
         }
       }
     currentPre ++;
   }
}
```

Table 2: StartElement handler for the StackEval tree-pattern evaluation algorithm.

traversal has finished, because at this time, all the descendants of the XML node for which the match was created have been traversed by the algorithm. Thus, we know for sure what parts of the queries could be matched in the descendants of the current node. The `endElement` handler plays two roles:

- prune out of the stacks those matches which satisfied the ancestor constraints but not the descendant constraints;
- close all `Match` objects corresponding to the XML element which has just finished (there may be several such matches, if several query nodes carry the same label). Closing these `Match`es is important as it is required in order for future tests made by the `startElement` handler to work.

```
class StackEval{ ...
   endElement(String localName){
     // we need to find out if the element ending now corresponded
     // to matches in some stacks
     // first, get the pre number of the element that ends now:
     int preOflastOpen = preOfOpenNodes.pop();
     // now look for Match objects having this pre number:
     for(s in rootStack.getDescendantStacks()){
       if (s.p.name == localName && s.top().status == open &&)
         s.top().pre == preOfLastOpen){
         // all descendants of this Match have been traversed by now.
         Match m = s.pop();
         // check if m has child matches for all children
         // of its pattern node
         for (pChild in s.p.getChildren()){
           // pChild is a child of the query node for which m was created
           if (m.children.get(pChild) == null){
             // m lacks a child Match for the pattern node pChild
             // we remove m from its Stack, detach it from its parent etc.
             remove(m, s);
           }
         }
         m.close();
       }
     }
   }
}
```

Table 3: EndElement handler for the StackEval tree-pattern evaluation algorithm.

Instructions Based on the previous explanation:

1. Implement an evaluation algorithm for C-TP tree-patterns. At the end of the execution, the stacks should contain only those `Match` objects that participate to complete query answers.

2. Implement an algorithm that computes the result tuples of C-TP tree patterns, out of the stacks' content.

3 Extensions to richer tree-patterns

Once this is implemented, the reader might want to consider implementing the extensions previously outlined. For all these extensions, a single traversal of the document suffices.

More precisely, one can consider:

1. Extend the evaluation algorithm developed in (1.) at the end of the previous section to

"*" wildcards. For this, `Stack` objects are allowed to be created for *-labeled query tree nodes. Also the `startElement` and `endElement` handlers are adapted.

2. Extend the evaluation algorithm to optional nodes, by modifying the tests performed in `endElement` (looking for children which the `Match` should have) to avoid pruning a `Match` if only optional children are missing.

3. Extend the algorithm developed in (2.) to handle optional nodes, by filling in partial result tuples with nulls as necessary.

4. Extend the evaluation algorithm to support value predicates, by (*i*) implementing a handler for the `characters(...)` SAX method, in order to record the character data contained within an XML element and (*ii*) using it to compare the text values of XML elements for which `Match` objects are created, to the value predicates imposed in the query.

5. Extend the algorithm in (2.) to return subtrees and not only preorder numbers. The subtrees are represented using the standard XML syntax with opening/closing tags.