Department Software and Computer Technology
Faculty EEMCS
DELFT UNIVERSITY OF TECHNOLOGY

Assignment-1
(Tree Pattern Evaluation using SAX)
Web Data Management (IN4331)
2012-2013

Ioanna Jivet
Nidhi Singh

June 2, 2013

1 Basic Setup

For this exercise, we have used two SAX parsers, one which parses the input tree pattern and the other which parses the main XML document to look for the input tree pattern and find matches. We chose to model the tree-pattern query as an XML and not give it as a string for two main reasons. Firstly, we avoided parsing the string and extracting the pattern-tree from it. The parsing of the pattern-tree can be done using a SAX parser. And second, the way the input is processed does not influence the execution of the algorithm, so we preferred to focus on the algorithm rather than the input format.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<people>
    <person>
        <email optional ="true" marked="true">
            m@home
        </email>
        <wildcard>
            <last marked="true">
            </last>
        </wildcard>
    </person>
</people>
```

Figure 1: Pattern-tree XML file

As shown in 1 the input tree pattern should contain the tree pattern nodes as tags, attributes should be assigned to show if a node is *marked*, or has an *optional* edge between its parent.

- **marked**: takes boolean value *true/false*, default: false
- **optional**: takes boolean value *true/false*, default: false

As the XML tag names cant contain the * character to mark a wildcard node, we opted for a keyword wildcard to mark such nodes. The text within each tag is treated similar to the predicate values in *where* clause of a query. Matched objects are created based on this condition.

```
public class PatternNode {

    private String name;
    private String text;
    private boolean marked = false;         //if the node is to be part of the resulting tuple; default false
    private boolean optional = false;       //if there is an optional edge between this node and its parent(default:false)
    private boolean wildcard = false;       //if the node is a wild-card; default false
    private ArrayList<PatternNode> children = new ArrayList<PatternNode>();
```

Figure 2: Attributes of PatternNode objects

The pattern-tree is stored in a tree-like structure, where each node is a PatternNode object. 2 shows the attributes of this class. The names of the attributes are self-explanatory. The parsing of this XML file into the tree structure is implemented in InputParser.java.

2 Exercises

2.1 Implementation of an evaluation algorithm for C-TP tree-patterns

2.1.1 Entities

The entities of the basic algorithm are abstracted by TPEStack.java, PatterNode.java and Match.java. The nodes in the tree-pattern are represented by a PatternNode object. For each node, a TPEStack is created. The stack contains all the matches found for the corresponding node. There is a one-to-many relation between the TPEStack and the Match objects. Each Match objects contains a reference to its parent Match, the one is linked to, and its child Matches. These references translate into the relation between the nested XML nodes in the analyzed XML file.

2.1.2 TP algorithm

The algorithm is implemented in the StackEval class. By using the SAX parser, the class implements the ContentHandler interface. The parsing of the XML file is done by implementing the startElement(...) and endElement() methods. The nodes in the XML file are numbered. The opened tags are monitored through a stack which contains the number of the opened XML element. All the TPEStack are held in a list. They will be used during the execution.

- **startElement(String uri, String localName, String qName, Attributes attributes)** when encountering a new tag, the TPEStack list is searched for a TPEStack that corresponds to the node that matches the name of the current opening tag. Additionally, if there exists an open tag for the parent of this node in the tree-pattern, then the match is valid and added in TPEStack corresponding to the current PatternNode. This Match is also linked to parent Matches, to keep the tree structure inside the matches. Next, the attributes of the opened tag are checked, in the same manner. Finally, the numbering of the open tags is incremented and the current value is pushed in the open tags stack.
- **endElement(String uri, String localName, String qName)** when encountering an end tag, the algorithm searches in the list for a TPEStack that corresponds to the node that equals the name of the tag. Again, if there exists an open parent tag and there is a recorded match for this tag, the tag is closed, to keep it in the Match stack inside the TPEStack, but to skip it in following steps. Further pruning is done by checking if the closed Match has all the child Matches, by inspecting the list of children. If at least one child match is missing, the entire match is removed and detached from its parent match.

At the end of the execution, the matches are found in the TPEStack object corresponding to the root PatternNode of the tree-pattern.

2.2 Implementation of an algorithm that computes the result tuples of C-TP tree patterns

2.2.1 Tuples extraction algorithm

To compute the resulting tuples, the TPEStack associated with the root node of the tree-pattern is traversed. For each match in the stack, the tree that results when adding the children matches and their descendants is traversed depth-first recursively to obtain the numbers assigned to the matching nodes. In Printer.java, two collection methods are implemented: one for all the matching nodes in the tree-pattern, and one only for the marked nodes.

2.3 Extension to include wildcard “*” node

The wildcard node in the tree-pattern can be substituted for any node in the XML file. Both startElement() and endElement() methods are modified. When opening/closing an XML tag, the list of TPEStacks is searched not only for TPEStacks of the nodes whose name equals the name of the tag, but also for a TPEStack that is linked to a node named wildcard. This reasoning was explained in the Input paragraph of this section.

2.4 Extension to include optional nodes

The optional nodes are denoted by an attribute optional in the XML file of the tree-pattern and a field in the PatternNode object. The feature is dealt with in the endElement() method of the evaluation algorithm. Only the non-optional children of the match are checked. The optional child matches are ignored, since it doesnt matter if they exist or not.

2.5 support for predicate values

To implement this extension, during the SAX parsing of the document, the text between element tags is stored in a HashMap. The keys of the map are the numbers associated with each tag. Storing and retrieving the text is easy. The text is collected by implementing the characters(char[] ch, int start, int length) method and adding the found text in the HashMap, using as key the preorder number of the last open tag. Since the PatternNodes have a field "text" that store the value predicate, the only alteration to the algorithm that needs to be done is in the endElement(). Before checking for child matches, the algorithm checks if there exists a value predicate that needs to be matched and compares it to the text stored in the Map, at the entry set by the number of the tag.

2.6 Eextension to return subtrees and not only preorder numbers

The final extension refers to the formatting of the output: returning subtrees and not only preorder numbers of the tags. The algorithm follows the same implementation as the one for extracting the tuples. Instead of adding the preorder numbers, they are used to retrieve data from the HashMap that contains the collected text elements. This text is wrapped in the name of PatternNode, surrounded by angle brackets.

2.7 Sample Run

2.7.1 Input

- tree-pattern

```
<person>
    <email optional ="true" marked="true">
    </email>
    <name>
        <last marked="true">
        </last>
    </name>
</person>
```

Figure 3: Tree-Pattern XML

- the example from the book.

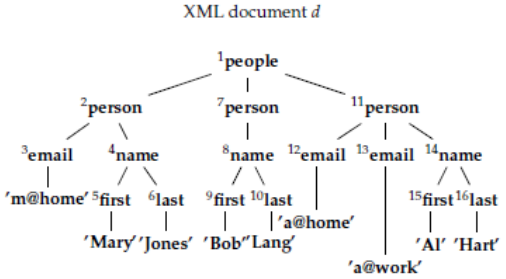

Figure 4: Tree-Pattern XML

2.7.2 Output

Figure 5 shows the obtained results

```
Full resulting tuples:
people  person  email   name    last
1       2       3       4       6
1       7       null    8       10
1       11      12      14      16
1       11      13      14      16

Marked resulting tuples:
email   last
3       6
null    10
12      16
13      16

Number of patterns found: 4

Marked resulting sub-trees:
email   last
<email>m@home</email> <last>Jones</last>
null    <last>Lang</last>
<email>a@home</email> <last>Hart</last>
<email>a@work</email> <last>Hart</last>
```

Figure 5: Result snippet