BlossomTree: Evaluating XPaths in FLWOR Expressions[*]

Ning Zhang
University of Waterloo
School of Computer Science
nzhang@uwaterloo.ca

Shishir K. Agrawal[†]
Indian Institute of Technology, Bombay
Department of Computer Science
shishk@cse.iitb.ac.in

M. Tamer Özsu
University of Waterloo
School of Computer Science
tozsu@uwaterloo.ca

Abstract

Efficient evaluation of path expressions has been studied extensively. However, evaluating more complex FLWOR expressions that contain multiple path expressions has not been well studied. In this paper, we propose a novel pattern matching approach, called BlossomTree, to evaluate a FLWOR expression that contains correlated path expressions. BlossomTree is a formalism to capture the semantics of the path expressions and their correlations. We propose a general algebraic framework (abstract data types and logical operators) to evaluate BlossomTree pattern matching that facilitates efficient evaluation and experimentation. We design efficient data structures and algorithms to implement the abstract data types and logical operators. Our experimental studies demonstrate that the BlossomTree approach can generate highly efficient query plans in different environments.

1. Introduction

In XML query languages, in particular XQuery, a path expression is arguably the most natural way to retrieve nodes from tree structured data, such as XML documents. In a more general setting, e.g., XQuery FLWOR (for-let-where-order-by-return) expressions, path expressions are usually used as building blocks for retrieving relevant nodes from XML documents. The intermediate results of these path expressions can be bound to variables, which are further referenced by other expressions.

The following FLWOR expression is such an example excerpted from XQuery Use Cases with minor modifications:

```
<bib> {
   for $book1 in doc("bib.xml")//book,
       $book2 in doc("bib.xml")//book
   let $aut1 := $book1/author
   let $aut2 := $book2/author
   where $book1 << $book2
       and not($book1/title = $book2/title)
       and deep-equal($aut1, $aut2)
   return
       <book-pair>
           { $book1/title }
           { $book2/title }
       </book-pair>
} </bib>
```

To evaluate such a query, we first formalize this FLWOR expression into a "BlossomTree" as shown in Figure 1. The BlossomTree is an annotated directed graph that captures all the semantics specified in the FLWOR expression. A vertex in the graph represents a tag-name and value constraints (no value constrains in this particular example) in a path expression. Variables can be bound to any vertex, in which case the vertex is called a *blossom*. An edge represents the (structural, value-based, or mixed) relationship between tree nodes that matches the two vertices. Among these edges, solid lines represent *tree edges* that are specified by path expressions. To distinguish different axes, we use regular solid lines to represent /-axes, and bold solid lines to represent //-axes. The dashed lines that connect two blossoms are *crossing edges*, which are usually generated by operators on variable bindings in the where-clause. To avoid clogging the graph, tree edges are not shown as directed, but their directions follow the hierarchical structure.

Given such a BlossomTree, the problem of evaluating FLWOR expressions can be reduced to matching the BlossomTree against an XML document. The focus of this paper is, therefore, to define a general algebraic framework for evaluating the BlossomTree and to develop optimization techniques based on this framework. The detailed version of this paper can be found in [1].

2. Evaluating BlossomTree

An algebraic framework includes a set of abstract data types and operators defined on these types.

[*]Research supported in part by grants from Communications and Information Technology Ontario and from the Natural Sciences and Engineering Research Council (NSERC) of Canada.

[†]This research was done while the author was visiting the University of Waterloo.

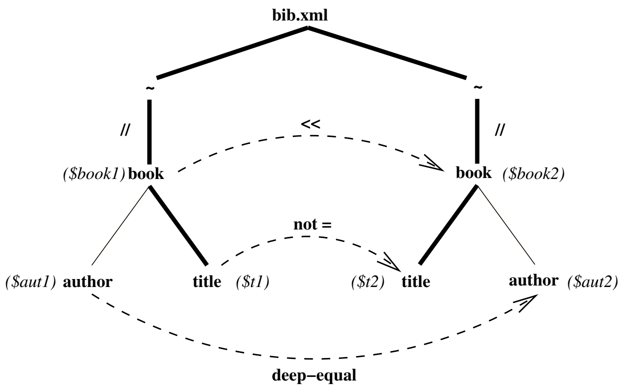

Figure 1. A BlossomTree

The basic idea of the design of the abstract data types is to define more "regular" data structures that isolate the irregular tree structures. This separation allows further operations to operate on the regular data structures only, thus the operators could be more efficiently implemented.

In our algebraic framework, the "more regular" data structure is NestedList, which is the output of NoK pattern matchings. Many operations, such as selection, projection and joins, can be applied to NestedList. Another abstract data type Env is generated when variables are bound to some specific values in the NestedList. The final XML document result is constructed from Env. The relationship between these abstract data types is illustrated in Figure 2.

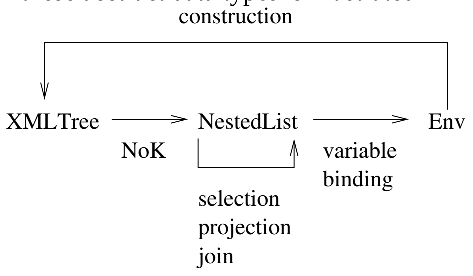

Figure 2. Abstract data types and operators

Consider the NoK Pattern Tree illustrated in Figure 3(a), where all nodes in the tree are returning nodes. To be able to reference the tree node, we first (arbitrarily) fix an order on the children, and assign each tree node a Dewey ID shown in the parenthesis. This artificial ordering does not affect the semantics of the pattern matching.

Figure 3(b) shows an example XML tree, where each t_i represents the i-th occurrence of tag t. The resulting tree is shown in Figure 3(c). The result of the match can be thought of as a special tree structure that conforms to the NoK pattern tree. For instance, all XML tree nodes (b1, b2, and b3) that are matched with pattern tree node 1.1 are grouped together, indicated by the box around the nodes. By grouping such nodes together, it is efficient to retrieve all nodes that are matched with a pattern tree node, given its Dewey ID. The tree edges in Figure 3(c) indicate which pairs of nodes satisfy the structural relationships specified by the NoK pattern tree. If x_i has an edge to y_m and, x_{i+1} has an edge to y_{m+k}, this implies that x_i can pair any of y_m up to y_{m+k-1} to form a match to the edge (x, y) in the NoK pattern tree. For instance, the tree edges between the b's and d's in Figure 3(c) indicate that b1 has no children in the match, b2 has two children d1 and d2, and b3 has one child d3 in the match.

The result tree is a compact representation of all mappings f from NoK Pattern Tree nodes to XML tree nodes. Conceptually, the result tree can be defined as an instance of an abstract data type, which we term NestedList.

Logical operators can be defined on NestedList. To be able to reference nodes, these operators take Dewey ID's as parameter. The signature of each of these operations is from a NestedList (in the case of Join, two NestedList's) to a NestedList. The semantics of each operator is as follows:

- Projection (π_{ID}): Navigate the NestedList according to the Dewey ID to a list of subtrees. The result is the concatenation of all the roots of the subtrees.
- Selection ($\sigma_{\varphi(ID)}$): First project on the Dewey ID, then evaluate the predicate φ on the projected list and remove items that return false.
- Join ($\bowtie_{\varphi(ID_1,ID_2)}$): First project the Dewey ID's on the corresponding two NestedList's, and then apply join predicate φ on the projected lists.

Note that the abstract data types and operators are defined at conceptual level, they can be implemented using different physical data structures and algorithms. We defined different algorithms for these operators. A particularly interesting operator is the join operator since it could be a blocking operator depending on the type of the join condition (descendant, document order, etc.). Based on different blocking properties, we developed pipelined joins and nested-loop joins for different types of join conditions. Our experiments show that different plans of the same query can result in very different performance. A cost model is crucial to choose the best plan.

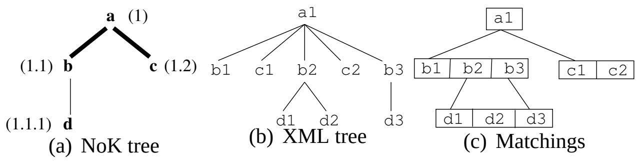

Figure 3. NoK Pattern Tree Pattern Matching against an XML tree

References

[1] N. Zhang, S. K. Agrawal, and M. T. Özsu. BlossomTree: Evaluating XPaths in FLWOR Expressions. Technical Report CS-2004-58, University of Waterloo, 2004. Available at http://db.uwaterloo.ca/~ddbms/publications/xml/TR-CS-2004-58.pdf.