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

Technical Report CS-2004-58 November 2004

University of Waterloo

[*]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.

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. These correlations include variable referencing, structural relationship, value-based relationship, or a mixture of structural and value-based relationship. 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 pattern matching approach can generate highly efficient query plans.

1 Introduction

XML data management, especially XML query processing, has attracted considerable research and development activity in the past few years. In XML query languages, in particular XQuery [6], a path expression is arguably the most natural way to retrieve nodes from tree structured data, such as XML documents. Path expressions allow us to traverse XML trees in different directions (termed "axes"), to filter tree nodes by specifying predicates on tag names and values, and to return tree nodes. By analogy, path expressions have a similar role in XML query processing as selection operations have in relational query processing.

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.

Example 1 Consider the following query, excerpted from XQuery Use Cases [8] (with minor modifications), that extracts, from a bibliography document, all pairs of distinct books that are written by the same list of authors:

```
<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>
```

There are 18 path expressions in this FLWOR expression, where two of them (both `doc()//book`'s) operate on the same XML document, while others operate on the variable bindings of these two path expressions (and the variable bindings thereof). The node comparison operator `<<` compares the document order of two nodes. The `deep-equal()` is a function that compares two subtrees. □

Since all the path expressions in the above FLWOR expression are related to each other, we call them *correlated path expressions*. More formally, two path expressions are correlated if they are related by variable references, structural relationships (e.g., `<<`), value-based relationships (e.g., `not =`), or value and structural mixed relationships (e.g., `deep-equal`). A straightforward approach to evaluating correlated path expressions in such a complex FLWOR expression is to follow the semantics of FLWOR expression and evaluate the path

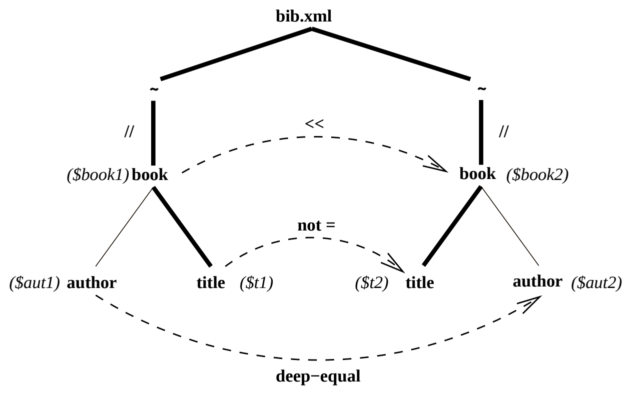


Figure 1: A BlossomTree for the query in Example 1

expressions for each iteration in the for-loop. However, this approach may be very inefficient, due to the redundancy during the loop and the costs of evaluating path expressions.

To tackle this problem, we first formalize this FLWOR expression into a "BlossomTree"[1] 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 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, while dashed lines represent *crossing edges* that are generated by operators on variables bindings in the where-clause. To avoid clogging the graph, tree edges are not shown as directed, but their directions follow the hierarchical structure. We shall give a formal definition and semantics for BlossomTree pattern matching in Section 3.1.

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.

Since BlossomTree is a generalization of pattern tree for path expressions, the evaluation of BlossomTree can be extended from previous research on pattern trees [17, 4, 20, 2, 7, 22]. There are basically three approaches to evaluating tree pattern matching: *navigational* approach, *join-based* approach and *hybrid* approach[2]. In this paper, we extend the hybrid approach [22] to a broader environment including FLWOR expressions. The objective of this research is to find out the pros and cons between the navigational and join-based approaches and try to tradeoff between them.

In summary, our contributions are as follows:

1. We propose a general algebraic framework (abstract data types and operators) for a substantial subset of FLWOR expressions.
2. We propose physical implementations of the abstract data type and its operators. In particular, data structures are proposed to support efficient implementation of operators.
3. The implementation of two major operators—generalized NoK pattern matching and structural joins—are discussed in detail. We propose a suite of join algorithms that exploit the properties of the input data.
4. We implement the techniques and conduct experiments based on our framework and join algorithms.

The remaining of the paper is organized as follows: we first introduce the background and related work in Section 2. The algebraic framework is presented in Section 3, where abstract data types are introduced in Section 3.2, and operators on these data types are presented in Section 3.3. The design of data structures and implementation of physical operators are introduced in Section 4. Implementation and experimental results are presented in Section 5. At last, we conclude in Section 6.

[1]The name BlossomTree is inspired by FLWOR and multiple returning nodes.

[2]We shall introduce these three approaches in detail in Section 2.

2 Background and Related Work

In this section, we briefly introduce the existing approaches to evaluating path expressions. Previous work on evaluating FLWOR expressions are also presented and compared with our approach.

2.1 Evaluating Path Expressions

The problem of evaluating path expressions against XML trees can be formalized as the *tree pattern matching* (TPM) problem (a.k.a. *twig query*): a path expression can be modeled as a pattern tree that specifies a set of constraints. The solution is to find all nodes in the XML tree that satisfy all the constraints, and return all those that match a "returning node" specified in the pattern tree.

Previous research on the evaluation and optimization of path expressions fall into three classes. A *navigational* approach traverses the tree structure and tests whether a tree node satisfies the constraints specified by the path expression [17, 4]. For example, the algorithm in [4] traverses the XML tree in pre-order (document order). This traversal is either through SAX event callbacks or is supported by the underlying storage system. A data structure called "work array" is established and maintained to keep track of which pattern tree nodes have been matched and which need to be matched for the incoming SAX events or XML tree nodes. This algorithm is very efficient when the document is non-recursive (i.e., an element could not be the descendant of itself). However, for recursive documents, the presence of '//' could result in very large memory requirement (in the order of the maximum number of recursive degree) [3].

Another approach is to first *select*, for each pattern tree node, a list of XML tree nodes that satisfy the constraints associated with it, and then pairwise *join* the lists based on their structural relationships (e.g., parent-child, ancestor-descendant, etc.) [20, 2, 7, 16]. Using proper labeling techniques [11, 9, 18], TPM can be evaluated reasonably efficiently by various join techniques (merge join [20], stack-based structural join [2], and holistic twig joins [7]). We call this the *join-based* approach. Join-based approach is scalable to large document and query size, by relying on the secondary storage to store intermediate results. Advanced algorithms (e.g., [2, 7]), which take advantage of the existence of tag-name indexes, can reduce the need to store intermediate results onto secondary storage, but the memory requirement is proportional to the maximum depth of the input document tree. A major disadvantage of the join-based approache is the update problem. Since it does not rely on navigation to discover the structural relationships between nodes, encodings of the structural relationship must be pre-computed and inserted into the indexes. This pre-computation can be thought of as a materialization process of the structural relationship. Thus, the join-based approach inherits the update problem associated with materialized views [5] (e.g., if a single element is inserted or deleted, the encodings of its subtree or all following nodes in the document may need to be recomputed).

We proposed a *hybrid* approach [22] to combine the advantages and avoid the disadvantages of the navigational and join-based approaches. The key idea is to, first, navigate through the XML documents to obtain intermediate results that contain structural information, and then join the intermediate results based on the structural information. In this way, we avoid the update problem since the structural information is obtained dynamically. To avoid the complexity of matching recursive documents in the navigational approach, we only navigate the tree using a subset of the path expression that will not result in a recursive matching. This methodology is implemented by first decomposing a general pattern tree into interconnected *next-of-kin* (NoK) pattern trees, which are special cases of a pattern tree that only consists of "/" and "following-sibling" axes. For each NoK pattern tree, a navigational NoK pattern matching operator can be evaluated highly efficiently using a single scan of the input, or index lookups. The efficiency is due to the fact that the recursive (or "global") axis "//" is excluded from the NoK pattern tree. The final result of the interconnected NoK pattern tree is then obtained by joining, based on their structural relationships, the intermediate results from the multiple NoK pattern matching.

We had shown previously that NoK pattern matching operator itself can be evaluated very efficiently [22], but optimizing the evaluation of more general expression containing multiple NoK expressions interconnected by global operations (e.g., `//`, `<<`) has not been studied. These cases occur when multiple NoK expressions exist in the same path expression or in different for-clauses. For example, the path expression `doc("bib.xml")/book[//author="Smith"]/title` retrieves the titles of the books that are written by "Smith". These two NoK patterns are interconnected by `//` relationship on `book` and `author` nodes. According to the definition of NoK patterns, this path expression should be partitioned into two NoK pattern trees, corresponding to `doc("bib.xml")/book/title` and `author[.="Smith"]`. Assuming that both NoK pattern matching operators are evaluated using sequential scan (i.e., for each XML tree node in document order, try to match the subtree starting from that node to the NoK pattern tree), we need to scan the input XML file twice, which is not I/O optimal. The FLWOR expression given in Example 1 is another example where multiple path expressions are connected by for- and let-clauses. Each of these path expressions can be decomposed into NoK pattern trees, which operate on the same document. Combining them as a whole tree to apply the hybrid approach may save significant I/O. This is the first issue we study in this paper.

The above examples motivate our first optimization technique: combining multiple NoK pattern matching operators into one scan whenever possible. Suppose a pattern tree is decomposed into two NoK pattern trees, each of which needs a scan of the (same) input XML tree. An intuitive observation is that the total number of scans of the input data can be reduced to one, if the intermediate results can be pipelined to a structural join operator without materializing them to secondary storage. We call this optimization technique *pipelined NoK* (corresponding to pipelined join technique). One of our objectives in this paper is, therefore, to analyze all types of structural joins[3] and determine what kinds of properties are required of the input streams in order to use the pipelined NoK technique. In fact, we will show that *only some types of structural joins* can use the pipelined NoK approach, while others cannot, no matter what kind of information is associated with the input nodes. However, pipelined NoK technique may not always be desirable since it requires more memory to cache intermediate results. This could be a serious problem when the input document and the path query is recursive. Therefore, we need to tradeoff between saving I/O and meeting the memory requirement.

2.2 Evaluating Path Expressions in the FLWOR Expression

Path expressions as the building blocks are often embedded in FLWOR expressions and constructor expressions (cf. Example 1). Evaluating them multiple times in a for-loop would greatly degrade the overall performance. Instead, if all the path expressions can be considered as one unit, and decomposed whenever needed, it could be much more efficient.

Previous research on evaluating path expressions in FLWOR expression can also be classified as navigational [12] and join-based approaches [10, 15]. These are natural extensions of the approaches to evaluating path expressions.

In the navigational approach [12], all path expressions are extracted from the FLWOR expression to form a single pattern tree. The pattern tree allows multiple nodes to be marked as "extraction nodes" (a.k.a., returning nodes), which correspond to variable bindings. The output of the pattern matching operator is a set of tuples, each column in the tuple corresponds to a different variable binding. Since the output is a relational table, this approach is particularly desirable when the XML query processor is built based upon relational database systems. However, since, in general, the result of the tree pattern matching is another tree whose nodes are mapped from the extracting nodes, relational tables (or flat tuples) are not space efficient in representing the tree structure. For example, consider a pattern tree with three extraction nodes "a", "b", and "c", where "a" is the parent of "b" and "c". If the XML tree has a subtree $\mathtt{a}_1$ with children $\mathtt{b}_1, \mathtt{c}_1, \mathtt{b}_2, \mathtt{c}_2$, where each $\mathtt{a}_i$ matches a, $\mathtt{b}_i$ matches b, and so on (throughout the paper, we use $\mathtt{a}_i$ to represent the i-th occurrence of element a), the resulting tuples are $(\mathtt{a}_1, \mathtt{b}_1, \mathtt{c}_1)$, $(\mathtt{a}_1, \mathtt{b}_1, \mathtt{c}_2)$, $(\mathtt{a}_1, \mathtt{b}_2, \mathtt{c}_1)$, and $(\mathtt{a}_1, \mathtt{b}_2, \mathtt{c}_2)$. This implies that the table contains redundancies due to the fact that its projection on b and c columns is actually a Cartesian product of the matches of all b and c.

On the other hand, the join-based approach [10, 15] treats every output of the pattern match as a tree. Each pattern tree node could be marked as a "tree logical class (TLC)", such that all matches to this pattern tree node can be retrieved by this TLC. Therefore, these TLC's can be thought of as references or indexes to the matched values of the "extraction nodes". In this sense, it is in the same role as the column names (or positions) in the tuple format in the navigational approach. Therefore, no structural relationships can be derived from the TLC's. Furthermore, the join-based approach also performs a structural join on each structural constraint, even though they can be answered more efficiently using navigation. When multiple path expressions are combined together, the number of structural joins will quickly become unmanageable for the optimizer to figure out an optimal join order.

[3]The types are defined based on the axes in path expressions [6].

In this paper, we extend the hybrid approach [22] for evaluating path expressions in a broader environment with FLWOR expression. The basic idea is twofold: (1) the path expressions are first combined together to form a BlossomTree to capture the semantics of the FLWOR expression. Then the BlossomTree is divided into NoK pattern trees as in [22]. The difference is that edge-cutting may happen not only on the edge labeled with global axes, but also value-based joins. (2) When a NoK pattern tree is constructed from multiple path expressions in a FLWOR expression, it could contain multiple returning nodes (blossoms). We define an abstract data type (NestedList) to represent the results of matching NoK Pattern Tree (which may contain multiple returning nodes) against XML document. NestedList is a generalization of flat list that allows an item be another list. Different from flat tuples or TLC references, each item in the NestedList has an ID (Dewey ID) that contains the structural relationships between matches from different pattern tree nodes. Furthermore, since NestedList can be thought of as trees, it also avoids the redundancy problem associated with the flat table.

In the following sections, we first introduce the abstract data type NestedList, and discuss how NestedList can be constructed from NoK Pattern Tree pattern matching. Then various operators on NestedList are presented.

3 An Algebraic Framework

In this section, we will introduce the formalism to capture the semantics of FLWOR expression. The abstract data type is presented in Section 3.2. The operators on the abstract data type are introduced in Section 3.3.

3.1 BlossomTree

The FLWOR expressions that we consider in this paper are a subset of that defined in XQuery [6], in that we only allow path expressions in the for-, let-, where-, order by- and return-clauses:

```
FLWOR   ::=   ( 'for' var 'in' Path | 'let' var ':=' Path )+
              ('where' Boolean)?
              ('order by' Path)?
              'return' Path
```

To capture the semantics of the restricted FLWOR expression, we introduce the formalism of BlossomTree.

Definition 1 (BlossomTree) A BlossomTree is an annotated directed graph that consists of interconnected pattern trees, at least one of which has a root node. Each vertex is annotated with a tag name and optional value constraints. In addition, each vertex can be associated with a variable, called *blossom*.

Each edge in the graph is annotated with a 2-tuple $\langle r, m \rangle$, where r is a relationship between the incident vertices, and m is a matching mode ("f" for mandatory and "l" for optional). □

Based on the definition, edges can be classified into *tree edges* and *crossing edges*. The relationships r associated with tree edges are always structural relationships (/, //, etc.), and the mode m could be either "f" or "l", representing whether the edges is contributed by a for-clause or let-clause, respectively. The relationships r associated with crossing edges can be either structural, value-based, or a mixture of structural and value-based relationships. In this case, the mode m could be "f" only.

For instance, the BlossomTree illustrated in Figure 1 has two "l" edges, represented by regular solid lines. The other tree edges are annotated with "f", represented by bold solid lines. The variables in the parentheses beside the nodes are bound to those nodes. Unless otherwise labeled, the default tree edge r relationship is /.

Taking a BlossomTree and an XML document, the semantics of the BlossomTree pattern matching can be defined using a mapping f from the BlossomTree nodes to XML tree nodes:

- The root of each pattern tree is mapped to the root element of the corresponding document.
- For any node u in the BlossomTree, $f(u)$ must satisfy the tag-name and value constraints associated with u.

- If two nodes u and v in the BlossomTree are connected by a directed edge (u, v) with annotation $\langle r, m\rangle$, then $f(u)$ and $f(v)$ must satisfy the relationship specified by r, i.e., $r(f(u), f(v)) = \mathsf{True}$.
- If two nodes u and v are connected by an "f" annotated edge, f is a valid mapping iff for each non-empty $f(u)$, there is a non-empty $f(v)$ satisfying the above conditions.
- If two nodes u and v are connected by an "l" annotated edge, f is a valid mapping iff $f(u)$ is non-empty, but $f(v)$ may be empty. In this case, $f(v)$ is defined to be an empty sequence if no XML tree nodes satisfy the above conditions.

Example 2 Consider matching BlossomTree in Figure 1 against the following XML document:

```
<bib>
  <book>
    <title> Maximum Security </title>
  </book>
  <book>
    <title> The Art of Computer Programming </title>
    <author>
      <last> Knuth </last>
      <first> Donald </first>
    </author>
  </book>
  <book>
    <title> Terrorist Hunter </title>
  </book>
  <book>
    <title> TeX Book </title>
    <author>
      <last> Knuth </last>
      <first> Donald </first>
    </author>
  </book>
</bib>
```

The output is:

```
<bib>
  <book-pair>
    <title> Maximum Security </title>
    <title> Terrorist Hunger </title>
  </book-pair>
  <book-pair>
    <title> The Art of Computer Programming </title>
    <title> TeX Book </title>
  </book-pair>
</bib>
```

The second book-pair being returned is obvious, since all the BlossomTree nodes are matched and all constraints are satisfied.

The first book-pair is in the output, because neither of these books have an `author` element, but the matching of them is optional (due to the "l" annotated edge connecting them), so both variables `$aut1` and `$aut2` are bound to empty sequences, which matches to each other by the definition of deep-equal() function (define in XQuery Functions and Operators [14]). □

Given such a BlossomTree, the hybrid approach first decomposes it into interconnected NoK Pattern Tree's. Each NoK Pattern Tree is evaluated by a navigational pattern matching operator. The intermediate results are then joined together using the join-based approach. It is straightforward to decompose a BlossomTree into interconnected NoK Pattern Trees. Algorithm 1 gives the pseudo-code to decompose a BlossomTree based on depth-first traversal. In the parameters, pt is a BlossomTree, S is a set containing the roots of the NoK Pattern Tree, and T is a set containing the non-root nodes in the current NoK Pattern Tree. Initially, S is set to be the set of all roots in pt, and T is set to be $\emptyset$.

Algorithm 1 Decomposing BlossomTree into NoK's

```
DECOMPOSE($pt$ : BlossomTree, $S, T$ : Set)
 1  while $S \neq \emptyset$
 2      do extract an item $u$ from $S$;
 3         $T \leftarrow \emptyset$;
 4         initialize $t$ as empty NoK tree;
 5         for each out-edge $(u, v)$ s.t. $v$ has not been visited
 6             do if the label of $(u, v)$ is a local axis
 7                   then set $v$ as visited;
 8                        put $v$ in $T$;
 9                        add $v$ and edge $(u, v)$ in t;
10                   else put $v$ in $S$;
11
12         DFS(t,S,T);
13         output $t$;

DFS($t$ : NoKBlossomTree, $S, T$ : Set)
 1  while $T \neq \emptyset$
 2      do extract an item $u$ from $T$;
 3         delete $u$ from $pt$;
 4         for each out-edge $(u, v)$ s.t. $v$ has not been visited
 5             do if the label of $(u, v)$ is a local axis
 6                   then set $v$ as visited;
 7                        put $v$ in $T$;
 8                        add $v$ and edge $(u, v)$ in t;
 9                   else put $v$ in $S$;
10         DFS(T,S,T);
```

3.2 Abstract Data Type

Before introducing how the pattern matching is performed on the NoK Pattern Tree, we first introduce the abstract data type (a.k.a. sort in algebra) that represents the output of pattern matching. Note that the abstract data type introduced here is at the conceptual level. It could be implemented by different concrete data structures (introduced in Section 4) to ensure efficiency.

Since the XML tree data model is very irregular and flexible, our basic idea of the design of the abstract data types is to define more "regular" data structures (thus operators based on it could be more efficiently implemented) that isolate the irregular tree structures. This separation allows further operations to operate on the regular data structures only.

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.

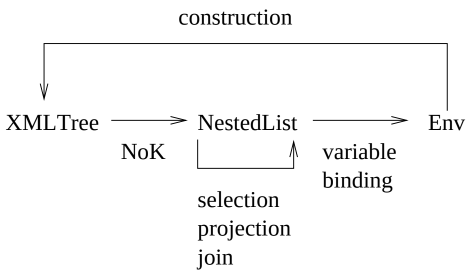
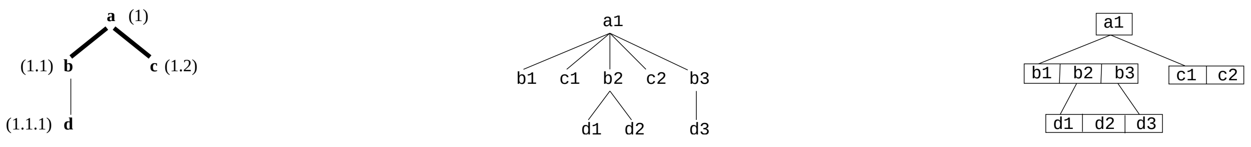


Figure 2: The data flow from one abstract data type to another

In this paper, we concentrate on the NestedList and operations on it. The Env abstract data type and how variables are bound are outside the scope of this paper. Interested readers are referred to [21] for a brief overview.

Before giving the formal definition, we shall give an example of how the NestedList is constructed.

Example 3 Consider the NoK Pattern Tree illustrated in Figure 3(a), where all nodes in the tree are returning nodes. Under the same convention, the bold edges are "f" annotated (mandatory matching) and regular edges are "l" annotated (optional matching). To be able to reference the tree node, we first (arbitrarily) fix an order on the children (since normal pattern tree is unordered), and assign each tree node a Dewey ID shown in the parenthesis. This artificial ordering does not affect the semantics of the pattern matching since it is never used to match any order (e.g., document order), but just used to conveniently reference nodes.



(a) NoK tree (b) XML tree (c) Matchings

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

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 (recall the semantics definition in Section 3.1). To derive a mapping from the result tree, simply choose one node from each "array" associated with each Dewey ID, and check whether any pair of nodes conform to the structural relationship specified by the NoK pattern tree.

Conceptually, the above "resulting tree" can be defined as an instance of an abstract data type, which we term NestedList. The NestedList abstract data type is a more restricted form and has some special features than the general nested list defined in languages such as Lisp. A NestedList can be constructed from a BlossomTree (or specially NoK Pattern Tree), where each returning node is given a Dewey ID. For instance, the initial NestedList constructed from the NoK Pattern Tree in Figure 3(a) is `(a(b(d))(c))`, where each '(' indicates a nesting, each ')' indicates a unnesting, and the characters (actually Dewey ID's of these characters) are placeholders for the matchings result to these characters. It is well-known that any tree can be represented by such a nested list form [13]. When the NoK pattern tree is matched with XML tree nodes, the placeholders are "filled out" with the actual nodes. To represent the "grouping" of multiple matchings with the same Dewey ID, a new notation "[]" is introduced. For example, the final resulting tree in Figure 3(c) is represented by the NestedList shown in Figure 4.

$$\texttt{(a1,}\underbrace{\lfloor\texttt{(b1,()),(b2,}\lfloor\texttt{(d1),(d2)}\rfloor\texttt{),(b3,(d3))}\rfloor}_{\texttt{b subtrees}}\texttt{,}\underbrace{\lfloor\texttt{(c1),(c2)}\rfloor}_{\texttt{c subtrees}}\texttt{)}$$

Figure 4: An example NestedList

In this notation, "[]" represents the grouping of subtrees under the same parent (e.g., grouping `b1`, `b2`, and `b3` subtrees, but not `d3` with `d1` and `d2`), and "()" represents nesting. Note that an empty sequence "()" is introduced to `b1`'s child in order to conform to the structure of NoK pattern tree.

Definition 2 (NestedList) The NestedList abstract data type is a nested list representation of an ordered tree structure that is leveraged by the grouping notation "[]". □

Note that the NestedList abstract data type is conceptual; one can use any data structure to implement it in order to guarantee good performance for certain operations on it. In Section 4, we shall introduce the physical data structures for NestedList to support efficient operations.

3.3 Operators on NestedList

In this subsection, we define logical operators on NestedList. These operators are very similar to relational operators, but working on a *sequence* of "nested tuples" instead of a *set* of "flat tuples". Instead of taking column names or positions as parameters in the flat tuple case, the operators on NestedList take Dewey ID's as parameter.

As shown in Figure 2, 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}): Unnest the NestedList according to the Dewey ID to a list of subtrees. The result is the concatenation of all the roots of the subtrees. For example, $\pi_{1.1}(t) = [\texttt{b1}, \texttt{b2}, \texttt{b3}]$, where t is the NestedList in Figure 4.
- Selection ($\sigma_{\varphi(ID)}$): First project on the Dewey ID, then evaluate the predicate φ on the projected list. If the predicate returns false, remove the corresponding item from the NestedList. If the resulting NestedList is not a valid match anymore (e.g., a mandatory node matches to empty sequence after the removal), return empty sequence; otherwise return the modified NestedList. For example, $\sigma_{position(1.1)=2} = [\texttt{b2}]$, where *position* is the list position function in path expression (e.g., `//book[2]`).

- Join ($\bowtie_{\varphi(ID_1,ID_2)}$): The join of two NestedList's is a combination of projection and selection. It first projects the Dewey ID's on the corresponding two NestedList's, and then apply join predicate φ on the projected lists. If the predicate φ returns true, the two NestedList's are combined into a single NestedList (see below for details); otherwise, return empty sequence.

This semantics can be easily extended to a sequence of NestedList's: simply concatenate the results on each NestedList in order.

The semantics for projection and selection are quite straightforward, however, the join semantics needs more elaboration. The problem is how to construct the resulting NestedList from two NestedList's. Consider the BlossomTree in Figure 1. Since matching to the whole BlossomTree should contain all the returning nodes (those with variable bindings), we should assign a Dewey ID to each returning node before decomposing it into interconnected NoK pattern trees. Since there are two root nodes in the BlossomTree, we introduce an artificial (super-)root that is the parent of both root nodes. The Dewey ID's can be assigned according to any ordering of the roots. Suppose that \$b1 is assigned 1.1, \$b2 is assigned 1.2, \$aut1 is assigned 1.1.1, \$t1 is assigned 1.1.2, \$t2 is assigned 1.2.1, and \$aut2 is assigned 1.2.2. The initial NestedList for each of the NoK pattern tree is of the form:

```
(($book1,($aut1),($t1)),($book2,($t2),($aut2)))
```

Each NoK pattern tree outputs a sequence of NestedList's that conform to the above format, although some results contain placeholders. When two sequences of NestedList's are joined, the placeholders are filled out in the result.

Example 4 Consider the BlossomTree in Figure 1 and the XML document in Example 2 as input, and suppose the execution plan is shown in Figure 5, where NoK_1 and NoK_2 are both `(book,(author,title))` with different variable bindings.

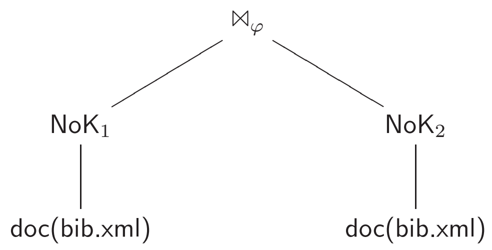


Figure 5: A query plan, where the join predicate $\varphi = (1.1.2 \neq 1.2.1) \wedge \textbf{deep-equal}(1.1.1, 1.2.2)$

The result of NoK_1 is of the form (assuming $\mathtt{b}_i$ represent the i-th occurrence of `book`, $\mathtt{a}_i$ represent the i-th occurrence of `author`, and $\mathtt{t}_i$ represent the i-th occurrence of `title`):

```
[((b1,(),(t1)), ((),())),
 ((b2,(a1),(t2)), ((),())),
 ((b3,(),(t3)), ((),())),
 ((b4,(a2),(t4)), ((),()))]
```

Each line above is a matching against NoK_1, and the empty sequences are placeholders. The result of NoK_2 is of the same form, but with placeholders in the position of NoK_1:

```
[(((),()), ((b1,(),(t1)))),
 (((),()), (b2,(a1),(t2))),
 (((),()), (b3,(),(t3))),
 (((),()), (b4,(a2),(t4)))]
```

The join operator takes the two sequences of NestedList's and the join predicate, and evaluates the predicate for each combination of the NestedList's from the two sequences. The final result is:

```
[((b1,(),(t1)),((b3,(),(t3)))),
 ((b2,(a1),(t2)),(b4,(a2),(t4)))]
```

This is the same result returned by Example 2. □

Given a NoK Pattern Tree and an XML tree, a *sequential scan* of the XML tree against the blossom tree is defined to be the process of scanning the XML tree node in document order. At each node, a NoK pattern matching operator is applied. The results of all matchings are concatenated into a sequence of NestedList's.

4 Physical Operators

In this section, we introduce how the operators defined in the previous section are implemented. Since selection and projection are rather straightforward, we concentrate on the NoK Pattern Tree pattern matching operator and various join operators.

4.1 NoK Pattern Tree Tree Pattern Matching

A NoK pattern tree could have multiple returning nodes, and optional edges ("l" annotated edges). As introduced in Section 3, we should give each returning node a Dewey ID indicating their structural relationships. This is achieved by extracting all returning nodes to form a "returning tree". In the returning tree, two nodes are connected by an edge if and only if they are the closest ancestor-descendant in the NoK tree. For example, in the NoK pattern tree, a has a child b and b has a child c, and only a and c are returning nodes, in the returning tree a and b are directly connected by an edge. Before introducing the NoK Pattern Tree pattern matching algorithm, we first introduce the data structure that implements NestedList abstract data type.

Given a Dewey ID, the data structure should support the following operations:

Unnesting a NestedList is one of the most common operations. The data structure should support efficient navigation (e.g., through pointer) between nesting levels.

Retrieving child by position index is another common operation. This operation usually follows unnesting and retrieve the i-th item in a sequence. For example, projection on Dewey ID 1.5.3 can be translated into the operations unnesting on the root, then retrieving the fifth child, then unnesting again, and last retrieving the third child.

Inserting a child to a list is frequently used in constructing the NestedList. Since the NestedList is dynamically constructed from depth-first traversal of the XML tree, it requires dynamically inserting an item into the sequence.

Based on these requirements, we design the data structure as shown in Figure 6. Each returning node is associated with a linked list (a is associated with list a1, etc.), where new element is inserted at the end. We say that this linked list is connected by *sibling pointers* (indicated by $\twoheadrightarrow$ arrows in Figure 6). If a returning node has k children in the BlossomTree, it includes an array of size k, containing pointers to the list associated with its children. We call these pointers *child pointers* (indicated by $\rightarrow$ arrows in Figure 6). In addition to these pointers, a pointer to the last child, a pointer to its parent, a pointer to the next node in document order are also added to facilitate efficient navigation. To avoid clogging the graph, we omit them in Figure 6, but readers should keep in mind that they are associated with each node in the data structure.

Given a returning tree, the resulting NestedList can be constructed as shown in Algorithm 2. Parameters p and q of the algorithm are NoK pattern tree nodes that are both returning nodes and p is a child of q. x is an XML tree node. Before invoking the algorithm, a prerequisite is that p matches with x.

This algorithm is an extension of the NoK pattern matching algorithm [22]: instead of returning only one list, it keeps a list of resulting nodes on each returning node and maintains the hierarchical structure between them. It traverses the XML tree in depth-first order (through First-Child and Following-Sibling in lines 8 and 19). During the traversal, it also recursively matches the NoK pattern tree node and builds up a list of matchings for each returning node. Each recursive call (lines 13 and 14) represents a matching of a NoK subtree against an XML subtree. In the algorithm, the sibling pointers and children pointers are established in lines 2 to 5. The frontier children mentioned in line 7 are the children of a NoK pattern tree

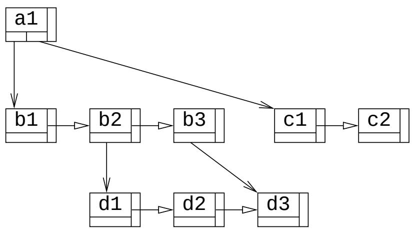


Figure 6: The concrete data structure for NestedList

Algorithm 2 NoK Pattern Tree Pattern Matching

```
NoK-Blossom($p, q$ : NoKNode, $x$ : XMLTreeNode)
 1  if $p$ is a returning node
 2     then append $x$ to the sibling list associated with $p$;
 3          $t \leftarrow$ last element of $q$;
 4          if $t$ has no child pointer
 5             then add a child pointer from $t$ to $x$;
 6             else  add $x$ to the last child of $t$;
 7  $S \leftarrow$ frontier children of $p$;
 8  $u \leftarrow$ First-Child($x$);
 9  repeat
10          for each $s \in S$ that matches $u$ with both tag
               name and value constraints
11                do
12                   if $p$ is a returning node
13                      then $b \leftarrow$ NoK-Blossom($s, p, u$);
14                      else  $b \leftarrow$ NoK-Blossom($s, q, u$);
15                   if $b =$ TRUE or $s$ is "l" annotated
16                      then $S \leftarrow S \setminus \{s\}$;
17                           delete $s$ and its incident arcs
                                from the pattern tree;
18                           insert new frontiers caused by
                                deleting $s$;
19          $u \leftarrow$ Following-Sibling($u$);
20     until $u =$ NIL or $S = \emptyset$
21  if $S \neq \emptyset$ and $p$ is a returning node
22     then remove all matches to $p$ associated with its parent
23          return FALSE;
24  return TRUE;
```

node that are candidate matches for the incoming XML tree node [22]. If after matching the whole subtree rooted at p, there are still unmatched pattern tree nodes (line 21), the partial results built up during the recursive call are removed (line 22).

Based on this data structure and algorithm, we shall prove that when we project on any Dewey ID, the resulting list associated with that DeweyID is in document order. We call any operator that has this property *order preserving*. This property is important for developing the join algorithms introduced in Section 4.2.

Theorem 1 (projection is order-preserving) *Suppose the sequential scan of an XML tree against a NoK Pattern Tree is s. Given any Dewey ID on the NoK pattern tree, the projection that simply returning the resulting list associated with that ID is in document order.*

Proof The proof has two steps: first, we prove that for any NestedList l that is the result of a non-recursive NoK pattern matching (i.e., no // connecting the NoK with the root), the projection on l is in document order. Secondly, we prove that for recursive matching, the list is still in document order.

The first step can be proved by the construction of the NestedList. Since the construction is by traversing the XML tree in document order, and a node is inserted into the result as soon as it is first discovered. Therefore, if two nodes are matched with the same pattern tree nodes, the node with lower document order must be inserted before the one with higher document order. Therefore, the projection on the NestedList on any Dewey ID is in document order.

The second step is also straightforward. Since a NoK pattern tree can have multiple matches during a recursive traversal of a subtree (in the case of recursive document), these matches can be interleaved together among the lists associated with each pattern tree node. By child pointers, different matches can be separated. When the NoK is recursively matched against a subtree, each matching node (in document order) is inserted into the list associated with its pattern tree node. These matches must in document order because they are inserted at the first time discover it in depth-first traversal (document order). ∎

4.2 Merging NoK operators and pipelined joins

As we have seen in Figure 5, there are situations where multiple NoK operators operate on the same XML file and the results are then joined together. There are two situations that an optimizer can exploit: (1) if both NoK operators use a sequential scan access method (e.g., when no tag-name or value-based indexes are available), we can save I/O by merging multiple NoK operators into one combined operator and using one scan only. (2) If the join connecting these two NoK operators has some special properties (e.g., the input is ordered by document order), we can use pipelined joins (merge-join style) to save the I/O for intermediate results. In this subsection, we shall present the two optimization techniques.

The merging of two NoK pattern trees proceeds as follows. For each NoK pattern tree, maintain a set of pattern tree nodes that are eligible for matching the incoming XML tree nodes (following the same notation in [22], we call these pattern tree nodes *frontier nodes*). When a new XML tree node arrives, it is matched to both sets of frontier nodes. This is in the same way that multiple Deterministic Finite Automata (DFA) are merged to a Nondeterministic Finite Automaton (NFA). This technique can be extended to merge arbitrary number of NoK pattern trees.

Given two sequence of NestedList's resulting from NoK operators, the join on DeweyID d_1 and d_2 is performed on the projections of d_1 and d_2 on the two NoK's. By Theorem 1, the results of the projections will preserve document order. Therefore, we can use any structural join algorithms (e.g., TwigStack [7]) that rely on the inputs being in document order.

However, some joins are not order-preserving. For example, the structural join on << operator is not order preserving. We shall give a counter example showing why it is not.

Example 5 Consider the example shown in Figure 1. A sub-query execution plan could be shown in Figure 7. Given the XML document as shown in Example 2, the resulting NestedList's are as follows:

```
[((b1,(),(t1)), (b2,(a1),(t2))),
 ((b1,(),(t1)), (b3,(),(t3))),
 ((b1,(),(t1)), (b4,(a2),(t4))),
 ((b2,(a1),(t2)), (b3,(),(t3))),
```

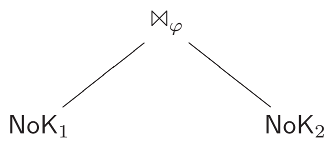


Figure 7: A Query Plan, where the join condition $\varphi = \$book1 << \$book2$

```
((b2,(a1),(t2)), (b4,(a2),(t4))),
((b3,(),(t3)), (b4,(a2),(t4)))]
```

The projection on the Dewey ID 1.2 over the above sequence of NestedList's is [b2, b3, b4, b3, b4, b4], which is not in document order. □

In fact, it is easy to show that some other structural joins, such as preceding, following, isnot, and all value-based joins are not document-order-preserving.

Nevertheless, the most commonly used //-joins are indeed document-order-preserving on non-recursive documents, which gives the nice property of being able to compose //-joins in pipelined fashion.

Theorem 2 (order-preserving on non-recursive documents) *If the NoK pattern trees connected by //-join operators are assigned Dewey ID globally (i.e., according to the BlossomTree, rather than individual NoK pattern tree), the result of //-joins preserves document order on non-recursive documents.*

Proof From Theorem 1, we know that the projections on each of the NoK pattern trees are order-preserving. We need to prove two cases: first, suppose l is a NestedList that matches the first NoK, and m_1, m_2 are the two consecutive NestedList's that join with l, since projection on NoK is order-preserving, m_1 and m_2 must be in document order, therefore, the join of l and m_1, m_2 must also be in document order.

Secondly, suppose l_1 and l_2 are two consecutive NestedList's that match the first NoK, and m_1 and m_2 are two consecutive NestedList's that matches the second NoK. Since projection on both NoK's are document-preserving, i.e., $l_1 < l_2$ and $m_1 < m_2$ in document order, l_1 and l_2 cannot both join with m_1 and m_2, i.e., the situation that (l_1, m_1), (l_1, m_2), (l_2, m_1), and (l_2, m_2) or (l_1, m_2) and (l_2, m_1) will never happen. This is because (1) the document is non-recursive, so l_i (or m_i) cannot match to the same pattern tree node, and (2) the tree structure prevents a node u being a descendant of a node v and v's sibling at the same time. Therefore, the projection of the result of the join is still in document order. ■

For //-join on non-recursive documents, we propose a pipelined join algorithm. As in the relational pipelined joins, the pipelined //-join and NoK pattern matching are defined using iterators: each time a getNext() function is called, a matching of the NoK pattern tree or join result is returned. The getNext() function of NoK pattern matching is based on Algorithm 2. Here, we only introduce the algorithm for the //-join operator.

```
GetNext(M, N : NoKTree; k, l : ID)
1  $m \leftarrow M.GetNext()$;
2  $n \leftarrow N.GetNext()$;
3  if $m = null \vee n = null$
4      then return null;
5  if Desc-or-Self($proj(m,k), proj(n,l)$) = TRUE
6      then return FILL(M,N);
7      else if $m << n$
8              then $m \leftarrow M.GetNext()$;
9              else $n \leftarrow N.GetNext()$;
10 goto 3;
```

Of these parameters, M and N are two pipelined NoK pattern matching operators, and k and l are two Dewey IDs on which the join is performed. This algorithm is in a merge-join fashion.

In this algorithm, the //-join GetNext function calls the NoK pattern matching operator's GetNext function (line 1 and 2). If one of them reaches the end (returning null), the join finishes as well. If m is not the ancestor of n or n itself, depending on which one has lower document order, another GetNext function call is invoked to the NoK pattern matching operator with lower document order (lines 7 to 9). If there are some nodes in the projection of m on DeweyID k that matches the projection of n on l, the resulting NestedList's are constructed by filling out the placeholders in one of the NestedList's and return it as a result.

The pipelined //-joins can save intermediate results being stored on secondary storage. However, if the input XML document is recursive, the order preserving property will not hold. Even if we modify the pipelined algorithm to cache more results to guarantee no results will be lost, the memory requirement for caching the intermediate results or candidates thereof could be large [3]. Therefore, in order to tradeoff between memory requirement and secondary storage access, the optimizer needs to have the knowledge of how recursive the input XML document is. For non-recursive or low degree of recursive document, the pipelined join algorithm (or its modification with caching capability) can be used. For highly recursive document, we propose a nested loop style join algorithm.

4.3 Nested Loop Joins

For those joins that are not order-preserving (e.g., <<-join, following-join, and isnot-join), a nested-loop join is in order. Since this join destroys the document order, we cannot compose other pipelined joins on top of the nested-loop join. For the joins that potentially return all pairs of combinations from two lists (essentially a Cartesian product), a straightforward naive nested loop join algorithm is unavoidable, so we omit it here.

For the nested loop joins on //-relationships, a special optimization technique can be applied. The basic idea is to exploit the fact if node u is an ancestor of v, then u's following sibling is not an ancestor of v. In the nested loop join algorithm, if NoK pattern tree T_1 connects another NoK pattern tree T_2 via a //-edge, T_1 is always on the left (outer) and T_2 is always on the right (inner). When the join operator get a nest matching from T_1, it will piggyback the range (p_1, p_2), where p_1 is the position of the XML tree node that matches with the joining node, and p_2 is the position of its nest sibling. The inner NoK T_2 can then only scan within this (p_1, p_2) range. We call this *bounded nested loop join* (BNLJ) algorithm.

5 Experimental Evaluation

categories	**recursive?**	**data set**	**size**	**#nodes**	**avg. dep.**	**max dep.**	**|tags|**	**|tree|**
Synthetic	Y	d1	69 MB	$1,212,548$	7	8	8	4.3 MB
	N	d2	17 MB	$403,201$	3	3	7	0.5 MB
		d3	30 MB	$620,604$	5	8	51	1.2 MB
Real	Y	d4	82 MB	$2,437,666$	8	36	250	5.3 MB
	N	d5	133 MB	$3,332,130$	3	6	35	8 MB

Table 1: Categories of testing data sets

The framework based on BlossomTree can be thought of as an algebra for the FLWOR expressions. As we have seen in the previous sections, a BlossomTree can be broken down into logical operators, each of which has different physical implementations (e.g., a structural join can be implemented with pipelined joins, nested loop joins, or TwigStack joins [7]). Each operator may have different performance on different data sets and queries. An optimizer is responsible for choosing an appropriate physical operator based on its knowledge of the system environment. Therefore, the purpose of our experiments is to find out in which situation a particular physical operator performs better than the others. In this paper, we focus on the join operator implementations.

To assess the effectiveness and efficiency of the proposed join operators, we conducted extensive experiments and compared them with the performance of existing systems or prototypes. Both the data and

Query	Category	Example query
Q1	hc	`/a/b//[c/d//e]`
Q2	hb	`/a//b[//c/d]//e/f`
Q3	mc	`//a//b//c`
Q4	mb	`//a/b[//c][//d][//e]`
Q5	lc	`//a//b`
Q6	lb	`//a[//b][//c]//e`

Table 2: Query categories

the queries are classified into categories so that we can test the efficiency of all approaches in different environments.

5.1 Experimental setting

We implemented the algorithms in Java with JDK 1.5. All the experiments were conducted on a PC with Pentium 4 2GHz CPU, 1GB RAM, and 200GB hard disk running Windows XP.

We conducted our experiments using both synthetic and real data sets (see Table 1). Both categories are further classified into recursive (an XML document is "recursive" if an element could appear as a descendant of itself) and non-recursive data sets. The reason that we include recursive data sets is to show the memory requirement for the online //-structural joins.

Of these synthetic data sets, d1 is generated from a recursive DTD, while d2 and d3 are chosen from XBench benchmark [19] (address and catalog, respectively). The real data sets d4 and d5 are selected from University of Washington XML Data Repository [1] (Treebank and dblp, respectively). Note that the recursive data sets also represent deep data, and the non-recursive data sets represent shallow (bushy) data.

When we select the testing queries, our concentration is on evaluating the performance of the tree navigation and join processing. The presence of value-based constraints is an orthogonal issue, therefore it is eliminated from our testing criteria. To cover the navigational and join processing aspects, we select queries based on the following properties:

Selectivity: A path expression returning a small number of results should be evaluated faster than those returning a large number of results. To evaluate whether our algorithm is sensitive to selectivity, we divided queries into three categories based on their selectivity: high (around 1% of total nodes), moderate (around 10% of total nodes), and low (around 50% of total nodes).

Topology: The topological properties of a pattern tree (which may be created from multiple path expressions) include how many NoK subtrees it includes, how many returning nodes each NoK subtree contains, whether these NoK subtrees form a chain or tree (with two or more branches), etc. To experimentally verify our join algorithms, we require that all the queries have at least two NoK subtrees and each NoK subtree has at least one returning nodes.

Combining these two criteria, we designed queries in six categories shown in Table 2. Each category is denoted by a string of length two, where each position denotes one of the above criterion. The character in each position stands for: low (l), moderate (m), or high (h) for selectivity; and chain (c), or branching (b) for topology. The tag names in the example queries are dummy and they should be replaced by appropriate tag names in different data sets.

5.2 Performance evaluation and analysis

We compared different join algorithms: pipelined (PL), bounded nested loop (NL), and TwigStack (TS) on different data sets. To compare with the industry strength product, we also tested the state-of-the-art native XML database system X-Hive/DB version 6.0 (XH). For each data set, we chose a representative path expression in each of the six categories.

Since the PL join is not order-preserving on recursive data sets, we only test NL, TS, and XH on these data sets. Not surprisingly, NL join has the worst performance on non-recursive data sets for all queries.

Therefore, we eliminate it from the experiment and only show PL, TS, and XH. The performance evaluation results for other systems are shown in Table 3. In this table, "DNF" denotes "did-not-finish" in 15 minutes. Each running time is the average over three executions.

file	sys.	Q1	Q2	Q3	Q4	Q5	Q6
d1	XH	5.1	24.32	6.63	17.59	7.54	49.58
	TS	0.12	4.19	13.18	20.54	10.26	11.69
	NL	10.41	30.92	DNF	DNF	114.03	DNF
d2	XH	4.46	8.52	3.22	14.31	3.22	13.14
	TS	0.94	2.23	0.96	1.01	1.38	2.25
	PL	0.92	1.20	0.99	1.24	1.02	1.22
d3	XH	3.54	3.95	3.47	3.86	3.94	14.21
	TS	0.95	0.51	0.44	0.77	0.25	0.40
	PL	1.30	1.24	0.62	0.62	0.90	1.11
d4	XH	15.87	15.23	14.59	15.22	16.42	75.85
	TS	5.337	7.61	5.06	5.99	5.46	3.96
	NL	DNF	DNF	22.38	DNF	DNF	DNF
d5	XH	33.56	127.95	33.11	138.36	34.26	140.42
	TS	4.97	0.18	2.62	4.03	0.25	3.71
	PL	3.86	3.89	3.91	3.87	3.99	4.10

Table 3: Running time (in sec) for X-Hive (XH), TwigStack (TS), and BlossomTree (BT)

Among these join algorithms, TwigStack is the most efficient algorithm in recursive documents. Since most of the axes in our test queries are //'s, it confirms the conclusion that TwigStack is optimal when all axes are //'s [7]. If the recursive data set is large, nested loop join is not efficient since it requires too many scans of the input. For non-recursive data sets, pipelined algorithm is comparable or even faster than TwigStack. Since TwigStack requires tag-name indexes, it is faster when the tag constraints in the query are selective. On the other hand, pipelined join algorithm does not rely on indexes, thus it resembles a sequential scan operator in relational databases. Therefore, pipelined algorithm is preferred in the stream context and in the case where no tag-name indexes are available. The optimizer needs to consider all these situations and makes the best choice of join algorithms based on its knowledge.

6 Conclusion and Future Work

FLWOR expressions are complex and usually containing multiple correlated path expressions. In this paper, we proposed a BlossomTree to combine these path expressions in a graph structure that contains all semantics specified in the FLWOR expression. A general algebraic framework is defined to evaluate the BlossomTree. Concrete data structures and algorithms are also designed to implement the abstract data types and logical operators defined in the framework. Our experimental results also show that the physical operators are highly efficient and the algebraic framework can generate multiple plans for the optimizer to choose.

Although we have efficient operators and access methods, the decisions on which operator to use requires a full-fledged optimizer. To choose an optimal plan automatically, the optimizer needs a cost model or similar mechanism. These will be topics of future work. Another future work is to investigate more expressions in XQuery, e.g., construction expressions, and develop a complete algebraic framework for the extended language.

References

[1] UW XML Data Repository. Available at http://www.cs.wa-shington.edu/research/xmldatasets/www/repository.html.

[2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. 18th Int. Conf. on Data Engineering*, pages 141–152, 2002.

[3] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. On the Memory Requirement of Evaluating XPath Queries over XML Streams. In *Proc. 23nd ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, 2004.

[4] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streming XPath Processing with Forward and Backword Axes. In *Proc. 19th Int. Conf. on Data Engineering*, 2003.

[5] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 61–71, 1986.

[6] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML Query Language. Available at http://www.w3.org/TR/xquery/.

[7] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 310–322, 2002.

[8] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie. XML Query Use Cases. Available at http://www.w3.org/TR/xmlquery-use-cases.

[9] Y. Chen, G. Mihaila, S. Padmanabhan, and R. Bordawekar. Labeling Your XML. preliminary version presented at CASCON'02, October 2002.

[10] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, and S. Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *Proc. 29th Int. Conf. on Very Large Data Bases*, 2003.

[11] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Trees. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 271–281, 2002.

[12] V. Josifovski, M. Fontoura, and A. Barta. Querying XML Streams. *The VLDB Journal*, 2004.

[13] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison Wesley, 3rd edition, 1997.

[14] A. Malhotra, J. Melton, J. Robie, and N. Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. Available at http://www.w3.org/TR/xquery-operators/.

[15] S. Pararizos, Y. Wu, L. V. Laksmannan, and H. Jagadish. Tree Logical Classes and Efficient Evaluation of XQuery. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2004.

[16] D. Shasha, J. T. L. Wang, and R. Giugno. Algorthmics and Applications of Tree and Graph Searching. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 39–53, 2002.

[17] J. Simeon and M. Fernandez. Available at http://www-db-out.bell-labs.com/galax/.

[18] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 204–215, 2002.

[19] B. B. Yao, M. T. Özsu, and N. Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. In *Proc. 20th Int. Conf. on Data Engineering*, 2004.

[20] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 425–436, 2001.

[21] N. Zhang. XML Query Processing and Optimization. In *20th Intl. Conf. on Data Engineering, Ph.D. Workshop*, 2004.

[22] N. Zhang, V. Kacholia, and M. T. Özsu. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *Proc. 20th Int. Conf. on Data Engineering*, pages 54 – 65, 2004.

Appendix A: Testing Queries

d1:
Q1(hc): //a//b4
Q2(hb): //a[//b2][//b1]//b3
Q3(mc): //a//c2/b1/c2/b1//c3
Q4 (mb): //a//c2//b1/c2[//c2[b1]]/b1//c3
Q5(lc): //b1//c2//b1
Q6(lb): //b1//c2[//c3]//b1

address (d2):
Q1 (hc). //addresses//street_address//name_of_state
Q2 (hb). //addresses[//zip_code][//country_id]
Q3 (mc). //addresses//street_address
Q4 (mb). //address[//name_of_state][//zip_code]//street_address
Q5 (lc). //address[//street_address]
Q6 (lb). //address[//street_address][//zip_code][//name_of_city]

catalog (d3):
Q1(hc): //item/attributes//length
Q2(hb): //item/title[//author/contact_information//street_address]
Q3(mc): //publisher//street_information//street_address
Q4(mb): //publisher[//mailing_address]//street_address
Q5(lc): //author//mailing_address//street_address
Q6(lb): //author[date_of_birth][//last_name]//street_address

treebank_e (d4):
Q1(hc)://VP//VP/NP//PP/PP
Q2(hb): //VP[VP]//VP[PP]/NP[PP]/NN
Q3(mc): //VP/VP/NP//NN
Q4(mb): //VP[VP]//VP/NP//NN
Q5(lc): //VP//VP/NP//PP/IN
Q6(lb): //VP[//NP][//VB]//JJ

dblp (d5):
Q1: //phdthesis//author
Q2. //phdthesis[//author][//school]
Q3. //www[//url]
Q4. //www[//editor][//title][//year]
Q5. //proceedings[//editor]
Q6. //proceedings[//editor][//year][//url]