Processing Recursive XQuery over XML Streams: The Raindrop Approach

Mingzhu Wei, Ming Li, Elke A. Rundensteiner and Murali Mani
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609, USA
{samanwei|minglee|rundenst|mmani}@cs.wpi.edu

***Abstract*— XML stream applications bring the challenge of efficiently processing queries on sequentially accessible token-based data. For efficient processing of queries, we need to ensure that memory usage stays low. This in turn requires that we avoid holding data in the query buffer, by outputting it at the earliest possible time. In this paper, we propose a new class of stream algebra operators for efficient recursive XQuery stream processing. In particular we propose two strategies for implementing structural joins: (a) the just-in-time structural join strategy efficiently processes joins as long as the input XML substreams are non-recursive and (b) the recursive structural join strategy supports structural joins over recursive XML substreams, however at an added cost of tuple-level ID-comparisons. Both structural join strategies are complemented by an automata-driven invocation mechanism that triggers the execution of the join at the first possible moment upon recognizing the end of the targeted input stream subelement. Further, we design this structural join operator itself to be context-aware. The operator is capable of at run-time switching from the efficient just-in-time join strategy for elements that are recognized to be non-recursive to the more powerful id-based structural join strategy for elements that are identified to be recursive. In addition, depending on whether the query is recursive, we will generate the plan with cheaper operators whenever possible. We incorporate the proposed techniques into the Raindrop stream engine. We also report on experimental studies we conducted using ToXgene that show that our techniques brings significant performance improvement.**

I. Motivation

XML has been widely accepted as the standard data representation for information exchange on the web. XML stream systems in particular have attracted more interest recently [4], [7], [8], [11], [9], [13] because of its wide range of applications such as sensor networking, online auctions, etc. XML data that appears in XML documents or XML streams frequently tend to be recursive. In the study of [2], 35 of 60 analyzed DTDs were recursive which shows that recursive XML schemas are very common in real world applications. Furthermore, queries may also be recursive as descendant axis(//) is often used in path expressions.

Among the current XML stream processing systems, some of them consider only XPath queries [8], [13], [5], while some others process XQuery over non-recursive data [11], [4]. YFilter [7] and Tukwila [9] can process XQuery over recursive data as well. However, in both YFilter and Tukwila, XQuery over recursive XML data is handled in a naive way by simply keeping all the context information. Therefore, they can not guarantee the joins are triggered at the earliest possible moment, thus leading to extra storage.

One solution for processing XQuery is to model the query semantics as a combination of automata and algebra [14], as is also done in [7], [9]. Let us first examine how the pattern retrieval is modeled as an automaton in Raindrop. Automata are naturally suited for matching path expressions over token sequences (a token can be a start tag, end tag or PCDATA item) since they were originally designed for matching regular expressions over alphabet sequences. Tokens that match the patterns are extracted from the stream and composed into XML element nodes, i.e., XML trees. These nodes are then wrapped into tuples and sent to an algebra-based query processor for further manipulation, such as filtering or restructuring.

As mentioned before, one of the key differences between Raindrop and systems such as YFilter and Tukwila is that in Raindrop, the joins are invoked at the earliest possible moment, thus optimizing storage and computation. Let us examine this feature of Raindrop with an example. Consider the XQuery Q1 given below. This query finds for each person, all its name descendants.

```
Q1:
for $a in stream(”persons”)//person
    return $a, $a//name
```

```
D1:
1 <person>
    2 <name>
       3  Jack
    4 </name>
    5 <children>
    6 </children>
7  </person>
8  <person>
    9 <name>
       10  Amy
    11 </name>
12 </person>
```

```
D2:
1 <person>
    2 <name>
       3  Jack
   4  </name>
   5 <children>
     6 <person>
        7 <name>
           8  Amy
        9 </name>
     10 </person>
   11 </children>
12 </person>
```

Fig. 1. Example XML Document Fragments. Document D1 is non-recursive, and document D2 is recursive.

Consider document D1 shown in Fig. 1. When we see the end tag for the first person (token 7), we have seen the entire content of this person element, along with all its name descendants. Now we can ”join” and output the person and all the name elements collected so far; then the buffer for storing person and name elements can be purged. We call this the just-in-time structural join [14]. This just-in-time structural join can be used when we see the end token of the second person element (token 12) as well.

The just-in-time structural join mentioned above performs simple cartesian product on person and name elements collected. However this simple cartesian product will not work when the data is recursive. For instance, consider document D2. Here the second person element (tokens 6 - 10) is a descendant of the first person element (tokens 1 - 12), we call such XML data as recursive. Note that the second name element (tokens 7 - 9) combines with both the person elements. Also the first person element and its descendant name elements need to be output before the second person element and its descendant name elements, based on the order restrictions imposed by XQuery. After the end tag of the first person element (token 12), we can join the two person elements with the ”appropriate” name elements, and output the results. Now the two person elements and the two name elements can be purged.

Since memory and CPU cost are both critical issues in XML stream processing, our goal is to optimize the storage and computation performance for handling XQuery, including recursive queries over XML streams. In this paper, we consider only recursive DTDs. If the schema is not recursive, it would be much easier to process recursive queries as we have studied in our previous work [14]. To process recursive queries combined with recursive schema, we have to tackle the following questions:

1. What should we do with the automata and algebra plan? Since our automata can retrieve patterns with descendant axis, it need not be changed. The intuitive idea is to change the algebra operators to let them cope with the recursive data.

2. How can we process recursive queries while keeping both the memory and computation cost as low as possible? Our goal is to make Raindrop process recursive query on recursive data as well as recursion-free data while keeping the cost minimal.

Our Approach

Our solution to deal with recursive XQuery includes the following:

(a) For recursive queries, we keep the ID information and level information to determine ancestor-descendant and parent-child relationships.

(b) To process recursive XQuery, we have to use recursive structural join which is essentially ID-based structural join, this is more expensive both in memory and computation. To save the memory and computation, we want to switch to cheaper just-in-time structural join as soon as we know that the XML data fragment is non-recursive. Thus we design this structural join operator itself to be context-aware. This operator is capable of at run-time switching from the efficient just-in-time join strategy for elements that are recognized to be non-recursive to the more powerful recursive structural join strategy for elements that are identified to be recursive.

Our contributions include:

1) We propose a new class of stream algebra operators for efficient recursive XQuery stream processing. We discuss the details in Section III.
2) We propose a context-aware structural join which switches from the efficient just-in-time join strategy for elements that are recognized to be non-recursive to the more powerful id-based structural join strategy for recursive elements at run-time.
3) We have two modes for each operator: the cheaper recursion-free mode and the more expensive recursive mode. Our plan generation examines the query, and uses the cheaper recursion-free mode operators whenever possible.
4) Our experiments illustrate that our invocation of the structural join at the earliest possible moment, our context-aware structural join and our clever plan generation bring significant performance benefits over other approaches.

II. Background

Raindrop uses automata-based model for pattern retrieval on tokens and uses algebra plan, which consists of algebra operators to do operations on sets of tuples. The query processing in Raindrop can be decomposed into two phases. In the first phase, tokens are processed by an extended automata engine for pattern retrieval. When the tokens are scanned and recognized, they are passed to algebra operators. In the second phase, these algebra operators create objects from these tokens, organize them into tuples, and perform further operations on these tuples.

A. Retrieving Patterns Using Automata

Our automaton is based on a non-deterministic finite machine (NFA). It encodes the path expressions present in the query. For instance, the automaton corresponding to Query Q1 is shown in Fig. 2. Here states s2 and s4 are final states, corresponding to the two path expressions in Q1.

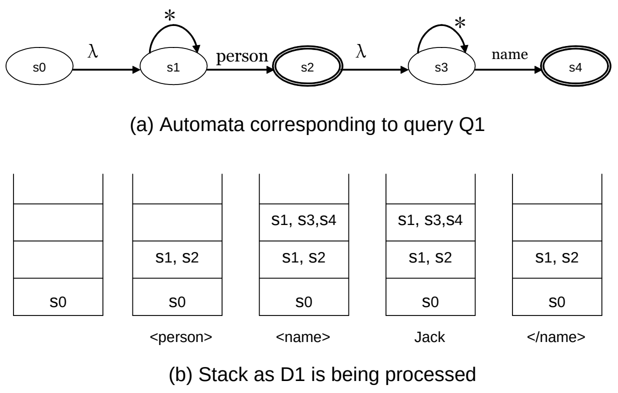


Fig. 2. (a) Automata (b) Stack

Our automata is augmented with a stack, which keeps track of the context of the tokens. Given a stream of tokens (the XML data), our automaton works as follows. Each final state in the automaton marks the end of a path expression. Given a current set of states S at the stack top, if the next token is a start tag, consider all the states to which any of the states in S can transit to for this token. These will form the new set of states, which are pushed on to the stack. If no state in S can transit to any state for the next token, then an empty set is pushed on to the stack. If the next token is an end tag, the current stack top is popped. The stack is then restored to the status before the matching start tag has been encountered. If the next token is a PCDATA item, this token is skipped.

Let us examine how patterns in document D1 in Fig. 1 are retrieved using the above automaton. Before the first token is seen, we have {s0} in the stack. When we see the start tag of person (token 1), we push {s1, s2}onto the stack. s2 is a final state; therefore, we have identified a pattern specified in the query Q1. In this case, the corresponding algebra operators are invoked, as will be discussed in the next subsection.

Now, we see the start tag of name (token 2). We push {s1, s3, s4} onto the stack. s4 is again a final state, and the corresponding algebra operators will be invoked. The next token (token 3) is a PCDATA item, in which case no action is taken. Then we see the end tag of name (token 4), now we pop the top of the stack. Any corresponding algebra operators associated with s4 for the end tag of name will also be invoked. This process continues, and we identify all the patterns.

B. Algebra plan

For any query, Raindrop generates an algebra plan that composes the tokens into tuples, and performs further operations on these tuples [14]. The algebra operators that are relevant to this paper are: Navigate, ExtractUnnest, ExtractNest and StructuralJoin operators. The description of each operator is shown in Fig.4. These algebra operators are invoked by the final states in the automaton. The algebra plan for query Q1 is shown in Fig. 3.

$Navigate_{path \rightarrow \$col}$ is invoked when an element that matches $path$ is identified by the automaton. This operator keeps track of the start and end of this element. It also notifies these events to its downstream Extract operators. For instance, $op1$ keeps track of the start and end tag of person elements, and notifies the Extract operator, $op4$ about these events. $ExtractUnnest_{\$col}$, when it is notified about the start tag from its upstream Navigate operator, starts collecting the tokens till it is notified about the end tag from its upstream Navigate operator. For instance, $op4$ will form one tuple for each person element. $ExtractNest_{\$col}$ is similar to $ExtractUnnest_{\$col}$, except that it groups all the $\$col$ into one tuple. For instance, $op3$ forms one tuple consisting of all the descendant name elements of a person. $StructuralJoin_{\$col}$ is invoked when an end tag of $\$col$ token is encountered by the "corresponding" Navigate operator. It combines (by performing cartesian product) the tuples from its branch operators. For instance, $op5$ is invoked whenever the end tag of person is seen by $op1$. It combines the person tuple from $op4$ and the set of names grouped into one tuple from $op3$, and outputs this result.

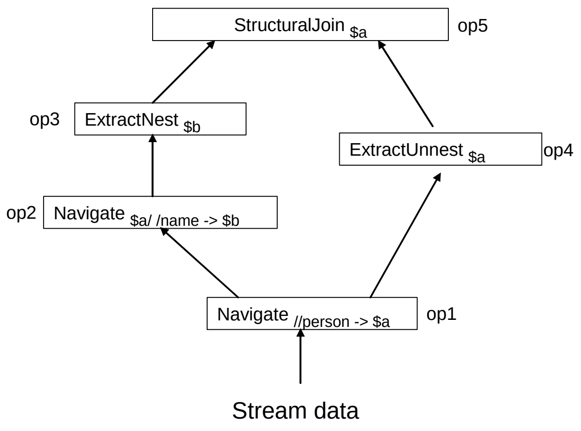
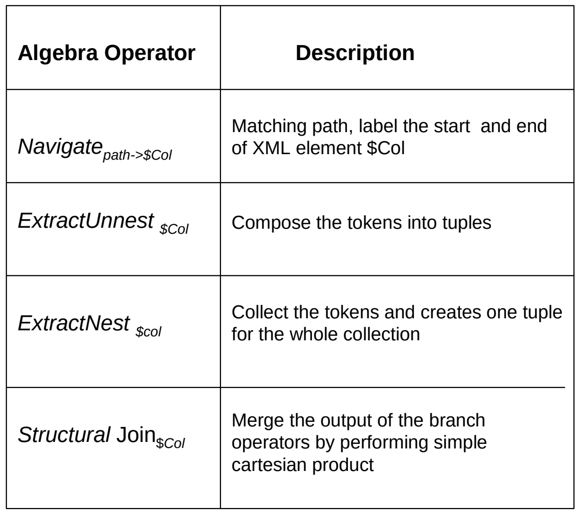


Fig. 3. Algebra plan corresponding to query Q1



Algebra Operator	Description
Navigate$_{path\text{->}\$Col}$	Matching path, label the start and end of XML element $Col
ExtractUnnest $_{\$Col}$	Compose the tokens into tuples
ExtractNest $_{\$col}$	Collect the tokens and creates one tuple for the whole collection
Structural Join$_{\$Col}$	Merge the output of the branch operators by performing simple cartesian product

Fig. 4. Algebra operators in Raindrop

C. Plan Execution

Let us examine how Raindrop combines the automata and the algebra plan to execute a query and obtain the results. Consider again, query Q1 executed on document D1. When the start tag of person (token 1) is seen, we push {s1, s2} onto the stack; s2 is a final state, therefore it invokes the operators associated with it – $op1$. The operator $op1$ in turn informs $op4$ and $op4$ starts collecting the tokens into a tuple. When the start tag of name (token 2) is seen, the states {s1, s3, s4} are pushed

	Query recursive	Query not recursive
Data recursive	Can't process	Generate correct output
Data not recursive	Generate correct output	Generate correct output

TABLE I
THE TECHNIQUES IN SECTION II CANNOT PROCESS RECURSIVE QUERIES ON RECURSIVE DATA

onto the stack. s4 is a final state; it invokes operator $op2$, which in turn invokes $op3$. $op3$ now starts collecting tokens.

When the end tag of name (token 4) is seen, we pop {s1, s3, s4} from the stack. $s4$ is a final state, it invokes $op2$, $op2$ informs $op3$ to stop collecting tokens. When the end tag of person (token 7) is seen, we pop {s1, s2} from the stack. $s2$ is a final state, it invokes $op1$, which informs $op4$ to stop collecting tokens. After this $op1$ invokes $op5$, which performs the join over the branch operators. In this case, there is one tuple from $op3$ and one tuple from $op4$ which are combined, and then output. Also the output buffers of $op3$ and $op4$ are cleared. The same process is continued for the second person element also. Note two features (a) the invocation of the structural join is done at the earliest possible time, thus ensuring buffers are cleaned up early (b) this structural join does a simple cartesian product, without any comparisons.

D. Issues for Recursive XML Data

The techniques mentioned in this section cannot be used for recursive XML data. Table I shows the cases that can be handled using the above techniques.

The techniques in this Section cannot process recursive queries on recursive data due to various reasons. For non-recursive data, the navigate operator invokes the structural join whenever the corresponding end tag is encountered. This does not work for recursive XML data, such as document D2. When we see the end tag of the second person (token 10), we have not seen the first person entirely. But the first person needs to be output before the second person according to the XQuery semantics. This means the Navigate operator will invoke the structural join only when the end tags for all the persons have been seen (that is, token 12 is seen).

Also for non-recursive data, the ExtractNest operator performs the grouping. This was possible because for D1 because any name element will join with at most one person element. This need not be true for recursive data; for instance, the second name element (tokens 7 - 9) joins with both the person elements. Therefore one solution, as what we will pursue in Raindrop, is that the ExtractNest does not peform the grouping; in stead the grouping is performed by the downstream structural join.

The structural join for non-recursive data did simple cartesian product of all its input branch operators. However, for recursive data such as D2, we have two person elements extracted by $op4$, and two name elements extracted by $op3$. We have to check the ancestor-descendant (or parent-child) relationships between these person and name elements. This requires additional information to be associated with the elements.

Note that the problems mentioned above happen when the query and the data are both recursive. In the next section, we will examine how Raindrop executes recursive queries over recursive data.

III. RECURSIVE-MODE OPERATORS

In this section, we will adapt the Raindrop operators so that they can process recursive queries over recursive data as well. We will first examine how we will associate additional information with each element. We will then investigate how each of the four algebra operators mentioned in Section II are modified to handle recursive data.

A. Associating IDs with elements

Each element is associated with a triple (startID, endID, level). Here, the startID of an element is given by the tokenID of the corresponding start tag, and its endID is given by the tokenID of the corresponding end tag. For instance, the startID of the first name element in D2 is 2, and the endID of this element is 4. The level of an element is the length of the path from the root to this element. For instance the level of the first name element is 1. This numbering scheme is similar to the DFS traversal numbering, as is also used in several other works [10].

Given two elements, and their corresponding triples, we can determine ancestor-descendant and parent-child relationships. For instance, consider the first person element in D2 whose triple is (1, 12, 0), and the first name element in D2 whose triple is (2, 4, 1). We can determine that the first name element is a descendant (also a child) of this person element.

B. Features of Recursive Navigate operators

The recursive Navigate operator functions differently from the non-recursive Navigate in several ways. First, the recursive Navigate operator keeps track of the triple for each corresponding element. These triples are kept in the order they arrive, which is the startID of the corresponding elements. For instance, consider document D2, and $op1$ in Fig. 3. Corresponding to the two person elements, $op1$ will keep the following two triples: $<(1,12,0),(6,10,2)>$.

Secondly, the non-recursive Navigate operator will call its structural join operator, whenever the end tag of the corresponding element is seen. But the recursive Navigate operator will call its structural join operator only when all the triples in this Navigate operator are complete, which means that we have seen the entire data for every one of these elements. For instance, when we see the end tag of the second person (token 10), $op1$ will have the following two triples: $<(1,\_,0),(6,10,2)>$; note that the first person element is not complete. Therefore $op5$ is not invoked. When the end tag of the first person (token 12) is seen, $op1$ will have the following two triples: $<(1,12,0),(6,10,2)>$. Both the person elements are complete, and therefore $op5$ will be invoked now.

Thirdly, the recursive Navigate operator needs to pass the triple information to the structural join in the order in which they are kept. Let us examine why the Navigate needs to pass this triple information by looking at an example query Q2.

```
Q2:
for $a in stream(”persons”)//person
    return $a//Mothername, $a//name
```

The algebraic plan for Q2 is similar to that shown in Fig. 3, except that $op4$ is now replaced by a Navigate and ExtractNest operators that extract the Mothernames for each person. Now the structural join when it receives the Mothernames and names for multiple person elements, it needs to know the person triples for determining which Mothernames and names join with which persons.

C. Features of Recursive ExtractUnnest operators

The non-recursive ExtractUnnest operator blindly extracts the tokens when invoked by the upstream Navigate operator, forms tuples from these tokens, and passes these tuples to its downstream Structural Join operator. The recursive ExtractUnnest operator, in addition to extracting the tokens into tuples, also adds the (startID, endID, level) information for every element to its corresponding tuple. For instance, consider query Q3 below.

```
Q3:
for $a in stream(”persons”)//person, $b in $a//name
    return $a, $b
```

The plan for Q3 will look similar to the plan in Fig. 3, except that $op4$ is replaced by $ExtractUnnest_{\$b}$. Now consider the two tuples corresponding to the two name elements in document D2 formed by $op4$. With the tuple for the first name element, $op4$ will add (2, 4, 1), and with the tuple for the second name element, $op4$ will add (7, 9, 3). This information will be used by $op5$ while performing the structural join.

D. Features of Recursive ExtractNest operators

The non-recursive ExtractNest operator groups all the tokens it has collected into one tuple. However, for recursive data D2, this is not feasible as the ExtractNest operator $op3$ stores the information only regarding the name elements. But the first name element (tokens 2 - 4) does not join with the second person element (6 - 10), and the second name element (tokens 7 - 9) joins with both the person elements. Therefore instead of $op3$ performing the grouping, Raindrop will move the grouping operation to the downstream structural join, $op5$ in this case.

The recursive ExtractNest in Raindrop is therefore similar to ExtractUnnest, that is it extracts tokens into tuples, adds the (startID, endID, level) information for every element to its corresponding tuple, and passes this information to the downstream structural join operator, which will perform the grouping, as will be mentioned below.

E. Features of Recursive StructuralJoin operators

As compared to the non-recursive StructuralJoin operator, the recursive StructuralJoin operator has to perform additional operations, including (a) ID-based comparison among its branch operators, and (b) grouping when its upstream operator is an ExtractNest operator. In this section, we will examine how the structural join is invoked, and its features.

1) Invoking Mechanism of Recursive Structural Join: For non-recursive data, $Structuraljoin_{\$person}$ ($op5$) in Fig. 3 is invoked whenever the end tag of $\$person$ is encountered. Such invoking mechanism brings problems when processing recursive data.

For example, when we process the data shown in document D2 in Figure1, the end tag of the second person (token 10) is encountered first which invokes $op5$. This structural join generates the output tuple composed of the second person element (tokens 6- 10) and the second name element (tokens 7 - 9). Then this person element and name element will be cleaned because they have been output by $op5$. When the end tag of the first person element (token 12) is encountered, $op5$ is invoked again. This time, the first person element (tokens 1 - 12) cannot join with the second name element because this name element has been deleted from the buffer. This happens for recursive data, because one name element can be descendant of multiple ancestor person elements. We do not want to delete the data which we will use later.

A second problem of this invoking mechanism is that the output does not conform to the stream order because the second person element is output first. To address this, suppose we have two person elements, one of which is a descendant of the other element, then we need to keep the data and invoke $op5$ only after the end of the outermost person element. Then we can guarantee that (a) we will not lose any data that will be needed later, and (b) the data is output in the correct order.

Now the question we face is how can we know that the end of the outermost person element has been reached? The structural join operator is invoked by the corresponding Navigate operator. In Raindrop, the Navigate operator will check whether the endID of all its triples have been filled. Only then it will invoke the structural join. For instance, $op1$ will invoke $op5$ only after seeing token 12. This ensures that the end of the outermost person element has been reached.

2) Algorithm for Recursive Structural Join: Consider $StructuralJoin_{\$col}$; let its branch operators be $B=\{bop_1, bop_2, \ldots, bop_n\}$. The algorithm for the recursive structural join is given below.

```
01  for each triple $t$ in $Navigate_{path \to \$col}$
02    for each branch operator $bop_i$
03      if $bop_i$ is $ExtractUnnest_{\$col}$
04        for each element $e$ in the output buffer of $bop_i$
05          if $t.startId = e.startId$
06            add $e$ to output list $o_i$;
07      else if branch operator contains //
08        for each element $e$ in the output buffer of $bop_i$
09          if $t.startId < e.startId$ and $t.endId > e.endId$
10            add $e$ to output list $o_i$;
11      else if branch operator does not contains //
12        for each element $e$ in the output buffer of $bop_i$
13          if $t.startId < e.startId$ and $t.endId > e.endId$
              and $e.level$ = $t.level + 1$
14            add $e$ to output list $o_i$;
15      if $bop_i$ is $ExtractNest_{\$col1}$
16        group $o_i$ to form one tuple;
17    perform cartesian product $O_t = \{o_1 \times o_2 \times \ldots \times o_n\}$;
      // $O_t$ is the result for the current triple.
18    add the tuples in $O_t$ to the output;
```

The structural join is invoked by its corresponding Navigate operator. This Navigate operator has one or more complete triples at this point. The structural join iterates over this set of triples (line 01 – 18). For each triple t, it goes through all the elements in every one of its branch operators (line 02 – 16). If the branch operator bop_i extracts the same element as the navigate, simply find the element corresponding to this triple t by performing ID comparison, and add it to the output list o_i for operator bop_i (line 03 – 06). Otherwise, we check whether bop_i contains // in its corresponding path expression. If it contains //, it means we need to determine ancestor-descendant relationship between t and every element e in the output buffer of bop_i by performing ID comparison. The descendants will be added to the output list o_i (line 08 – 10). If bop_i does not contain //, it means we need to determine parent-child relationship between t and every element e in the output buffer of bop_i. The children of t will be added to the output list o_i (line 12 – 14).

After the above ID comparison, o_i contains the set of elements corresponding to the current triple, for this branch operator. Now for the ExtractNest operators among the branch operators, we need to group all the elements in o_i to form one tuple (line 15 – 16). Now, we have obtained the set of elements corresponding to the current triple. We can simply perform the cartesian product of these o_i's and this generates the output for the current triple t. We continue to iterate over the remaining triples. After we have iterated over all the triples, the output buffers of all the branch operators are purged. Observe that the output tuples generated by the structural join are in the correct order; also the data is cleaned at the earliest possible time.

IV. Further Optimization

A. Context-aware Structural Join operators

As discussed before, recursive structural join needs to perform ID comparison and thus is more expensive than the just-in-time structural join which only performs cartesian product. To reduce the cost of the overall plan, we want to use the cheap structural join whenever possible.

We can determine whether the current data fragment is recursive or not by checking the number of triples passed to the structural join from the Navigate operator. If there is only one triple buffered in the Navigate operator, it implies that the current element is not recursive. In this case, structural join will execute the just-in-time structural join which has no ID comparison and thus is faster. If there are more than one triple stored in the Navigate operator, we have to perform recursive structural join. We incorporate this execution approach as a new type of structural join, which we call *context-aware structural join*. Context-aware structural join is capable of at run-time switching from the efficient just-in-time join strategy for elements that are recognized to be non-recursive to the more powerful id-based structural join strategy for elements that are identified to be recursive. The execution process of context-aware structural join is shown in Fig. 5.

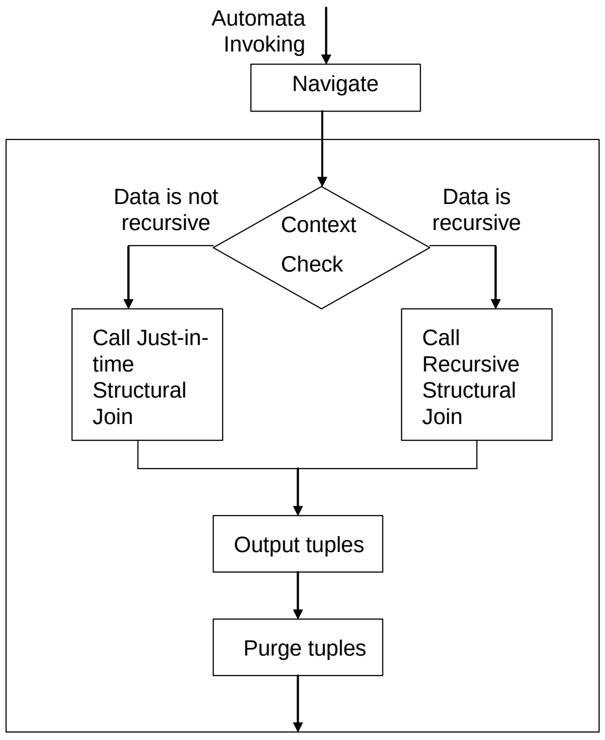


Fig. 5. Execution Process of Context-aware Structural Join

From the figure, see that when an appropriate end tag is recognized by the automaton, it informs the Navigate operator. This Navigate operator in turn invokes the context-aware structural join operator, if all the triples in the Navigate operator are filled. The context-aware structural join first checks whether this data fragment is recursive or not, by checking whether there are multiple triples in the Navigate operator. This is shown as the Context Check step in Fig. 5. If the data is not recursive, then the just-in-time structural join strategy is called; otherwise, the recursive structural join strategy is called. These results are then output, and the joined tuples are purged.

B. Optimizing Generated Plan based on Query

We studied above how the context-aware structural join chooses an efficient join strategy based on the data fragments at run-time. Now, we will examine how based on the query, we can generate more efficient plans.

For every operator, we have two modes: a *recursive mode*, and a *recursion-free mode*. A recursive mode Navigate operator keeps track of the (startID, endID, level) triple for each element, whereas a recursion-free mode Navigate operator does not keep any triple information. A recursive mode Extract operator adds the (startID, endID, level) triple to every element it extracts; however, a recursion-free mode Extract operator only collects the tokens into tuples without the triple information. A recursive mode structural join is the context-aware structural join that performs ID comparison as needed; whereas a recursion-free mode structural join uses the just-in-time structural join with no ID comparison. The recursive mode operators obviously require more memory, and perform more computation than the corresponding recursion-free mode operators. Therefore, we would like to use the recursion-free mode operators whenever possible.

Such plans are generated as follows. During plan generation, we check whether a structural join corresponds to a path expression with //. If so, this structural join, as well as all its descendant operators in the plan are instantiated as recursive mode operators. Consider query Q1, the plan generated for this is shown in Fig. 3, where every operator is a recursive mode operator. However, suppose the query is modified to be recursion-free as shown in Q4 below. Now, the plan generated is similar to Fig. 3, however, every operator will be a recursion-free mode operator.

```
Q4:
for $a in stream(”persons”)/person
     return $a, $a/name
```

C. Plans with Multiple Structural Joins

For a given XQuery, we might come up with a plan with multiple structural joins, where a structural join operator might have other structural join operators as its descendants. If the data is recursive, then the upstream structural join operator needs to pass the ID information to its downstream structural join operator. Let us consider this with an example query Q5 as shown below.

```
Q5:
for $a in stream(”s”)//a
  return {
    for $b in $a//b
      return {
        for $c in $b//c
          return { $c//d, $c//e },
        $b//f },
    $a//g }
```

The plan for query Q5 is shown in Fig. 6. For plans with multiple structural joins, the upstream structural join operator $StructuralJoin_{\$col}$ appends the (startID, endID, level) triple information of the corresponding $\$col$ to each output tuple.

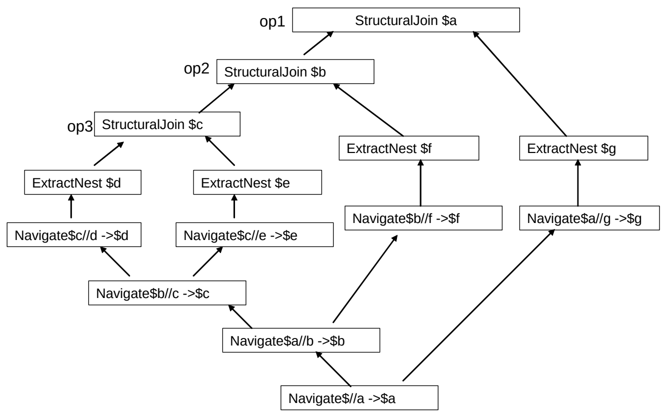


Fig. 6. Plan for Q5 with Multiple Structural Join Operators

For instance, consider $op3$ in Fig. 6. It is a context-aware structural join, which produces one tuple corresponding to each $c, with the grouped $c//d and $c//e tuples. To this tuple, it also appends the (startID, endID, level) corresponding to this $c, and passes this to $op2$. Similarly, $op2$ appends the triple corresponding to $b to each of its output tuples.

1) Optimizing Plan Generation with Multiple Structural Joins: During plan generation, determining whether an operator is a recursive mode operator or a recursion-free mode operator when we have multiple structural joins is done as follows. We traverse the query plan operators in a top-down manner. When we encounter a structural join operator that corresponds to a path expression with //, we instantiate this structural join operator and all its descendants as recursive mode operators. Other operators are instantiated as recursion-free mode operators.

V. Related work

[12], [3] evaluate XQuery expressions over XML streams using a two-phase approach which separates the task of pattern retrieval from the actual task of query evaluation, such as filtering, joining or restructuring. Recursion could be handled in such an approach in the second phase, i.e., using the well-studied algebraic query evaluation methods typically applied to (static, i.e., non-stream) XML data. However, such a two-phase solution does not fit well with the requirements of stream processing applications which require continuous query evaluation and immediate data purging to preserve memory.

[5] proposes to apply an encoding scheme in order to handle recursion. However, only XPath expressions have been considered in their work. In XQuery, careful synchronization between the pattern retrieval and query execution would need to be designed in order to achieve acceptable performance for recursion handling.

[11] uses a transducer model for evaluating XQuery over streaming XML. They do not consider recursive schemas. Transducers are simply FSA, augmented with buffers where you can store data or output data to. However, FSA without stack are not sufficient for handling recursion.

YFilter [15] follows the algebraic paradigm, introducing node-label trees for on-the-fly XQuery evaluation. Their main focus is on the evaluation of multiple queries. However, the approach of YFilter is not optimal for handling streaming XML input. This is because the proposed whole-path loading approach causes storage redundancy and extra join cost. Also, the evaluation strategy for the join operator cannot guarantee that the structural join will always be executed at the first possible moment – which is a desirable property to aid us in purging the buffer immediately and avoiding output delay.

The goal of [4] is to minimize the buffer size. Again, recursion handling is not considered in their work. Their focus is on an orthogonal issue; they study when a token can be output at the earliest. The Raindrop can incorporate their techniques to generate fast output as well.

Another algorithm similar to our recursive structural join strategy is the tree-merge join algorithm in [1]. However, they do not consider the streaming scenario; therefore the performance benefits of invoking the structural join at the earliest possible time is not their focus. Another algorithm mentioned in [1] is the stack-tree join algorithm. Here, they use two lists: an ancestor list and a descendent list, and they use a stack to keep the ancestor-descendent relationship among the elements in the ancestor list. So whenever an element at the top of the stack matches a descendent candidate, they can conclude that all the elements in the stack can also match that descendent candidate. But these generated tuples can not be output immediately because they do not conform to the document order. To solve this problem, this algorithm uses two extra lists for every node in the stack: one is the self-list which is the list of result elements from the join of this node with appropriate descendent elements. The second list, the inherit-list, is a list of join results for this node obtained from the join results of its descendants among the ancestor list elements. The disadvantages of the stack-tree join algorithm is that since the inherit-list will be appended to self-list whenever an element in the stack is popped, a large storage space is needed.

VI. Experiment Results

We use ToXgene[6], an XML data generator, to generate XML documents. All the experiments are run on a 2.8GHz Pentium processor with 512MB memory. We perform three sets of experiments. The first one shows that when joins are invoked at the earliest possible time, we require less memory for query processing. The second set of experiments shows that context-aware structural join is more efficient than always using recursive structural join. The final set of experiments shows the benefit of our plan generation method, where we use recursion-free mode operators whenever possible.

A. Advantages of Early Invocation of Structural Join

In this set of experiments, we study the memory usage when we invoke structural join at different times. We use query Q1, where the earliest time the structural join can be invoked is when we see the end tag for an outermost person. We measure the memory usage by counting the number of tokens we need to hold in the buffer before we invoke structural join. Once structural join is performed, the joined tuples are purged from the buffer. Fig. 7 shows the average number of tokens stored in the buffer (defined by the expression below) for five cases: zero-token delay (i.e., when structural join is invoked at the earliest possible time); one-token delay (i.e., structural join is invoked one token later than the earliest possible time); two-token delay; three-token delay; and four-token delay. Note that four-token delay causes about 50% more tokens to be stored than the zero-token delay.

The definition of the average number of tokens stored in the buffer is given by:

$$\text{Average number of tokens buffered} = \frac{\Sigma_{i=1}^{n} b_i}{n}$$

Here, b_i is the number of tokens stored in the buffer after we see token i; n is the total number of tokens we process.

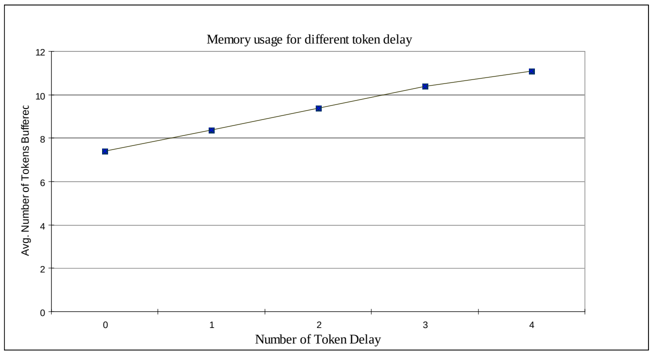


Fig. 7. The memory usage for different number token delay

Fig. 7 shows only the buffer saving when structural join is invoked at the earliest possible time, i.e., zero-token delay. Actually computation is also saved as fewer ID comparisons need to be performed when there is zero-token delay.

B. Efficiency of Context-Aware Structural Join

The experiments reported in this section illustrate the performance benefits of using the context-aware structural join rather than always using recursive structural join. Context-aware structural join uses just-in-time structural join strategy when it finds that this data fragment is not recursive; otherwise it uses recursive structural join strategy. For this set of experiments, we generate data sets, each of which has a size of about 30 MB, with varying percentage of recursive data from 20% to 100%. A data set with say 20% recursive data is generated as follows. We generate the recursive data portion of about 6 MB and the non-recursive data portion of about 24 MB separately using ToXgene; then we compose these two data portions into one XML file.

The query we use is query Q3. Q3 tries to find for every person element, its name descendants, and for each such name, it returns the person element, and this name element.

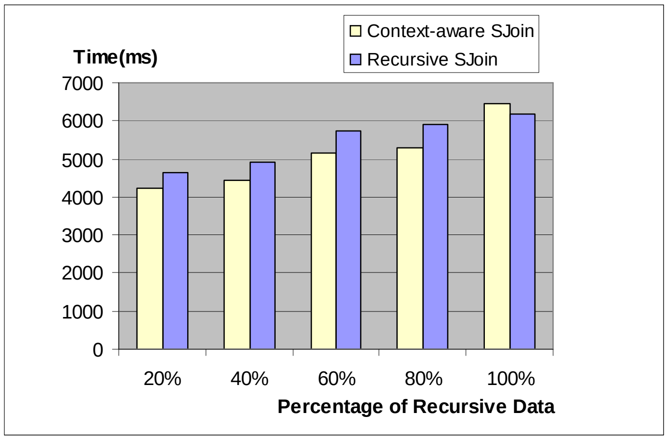


Fig. 8. Performance difference between context-aware structural join and recursive structural join with varying amount of recursive data

When 100% of the data is recursive, the context-aware structural join will always use the recursive structural join strategy, and hence has no extra benefit. Note that there is a small overhead, caused by the fact that the context-aware structural join has to check every time whether the data is recursive or not. However when the percentage of recursive data is less than 100%, there is always benefit in using context-aware structural join, as shown in Fig. 8.

C. Advantage of Using Recursion-Free Mode Operators

In this section, we study the benefits of our clever plan generation, where we analyze the query, and try to use as many recursion-free mode operators as possible. Recursion-free mode operators are more efficient than recursive mode operators even if they operate on the same data. The query we use for this set of experiments is Q6 given below.

```
Q6:
for $a in stream(”persons”)/root/person,
      $b in $a/name
      return $a, $b
```

When we analyze the query and see that the path expression corresponding to $a has no //, we would generate a plan where all operators would be recursion-free mode; for instance, we would have a just-in-time structural join on $a. If we had not performed this query analysis, we would have used recursive mode operators; for instance, we would have a context-aware structural join on $a. Fig. 9 shows the execution time difference between using recursion-free mode operators, and recursive mode operators for Q6 running on non-recursive data from 6 MB (that outputs 2K tuples) to 42 MB (that outputs 14K tuples). Observe that using recursion-free mode operators saves about 20% of the total execution time.

VII. Conclusion

We propose a new class of stream algebra operators for efficient recursive XQuery stream processing. In particular we propose two strategies for implementing structural joins: (a) the just-in-time structural join strategy efficiently processes joins when the XML substreams are non-recursive and (b) the recursive structural join strategy supports structural joins with ID-comparisons when the XML substreams are recursive. In this paper, we have proposed a context-aware structural join. This context-aware structural join uses the recursive structural join strategy when the data is recursive. When the data is not recursive, it switches to the cheaper just-in-time structural join strategy. Further, our structural join is invoked at the earliest possible time, leading to less memory usage. In addition, during our plan generation phase, we analyze the query and use the cheaper recursion-free mode operators whenever possible. Our experiments illustrate the performance gain achievable by each of the above techniques.

As part of our future work, we are currently investigating how to incorporate schema into our analysis. For instance, based on schema, we can generate plans with only operators for paths that exist and generate more recursion-free mode operators. Also the schema can be used for outputting tokens and invoking structural join earlier. In addition, we only consider forward axes in this paper. We can extend our work to incorporate backward axes in the future.

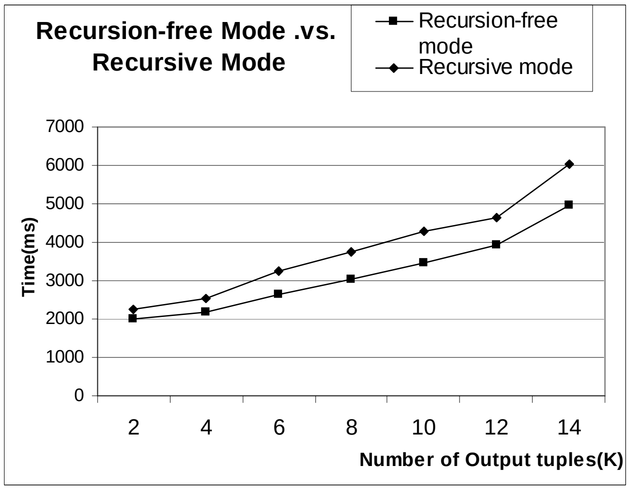


Fig. 9. Time comparison between recursion-free mode and recursive mode operators

References

[1] 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 *IEEE International Conference on Data Engineering (ICDE)*, page 141, Feb 2002.

[2] B.Choi. What are Real DTDs like. In *Proceedings of WebDB*, pages 43–48, 2002.

[3] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation- vs. index-based xml multi-query processing. In *IEEE International Conference on Data Engineering (ICDE)*, pages 139–150, 2003.

[4] C. Koch, S. Scherzinger, N. Scheweikardt and B. Stegmaier. FluxQuery: An Optimizing XQuery Processor for Streaming XML Data. In *International Conference on Very Large Data Bases (VLDB)*, pages 228–239, 2004.

[5] Y. Chen, G. A. Mihaila, S. B. Davidson, and S. Padmanabhan. EXPedite: A System for Encoded XML Processing. In *International Conference on Information and Knowledge Management (CIKM)*, number 108-117, 2004.

[6] D. Barbosa, A. Mendelzon, and J. Keenleyside et al. ToXgene: a Template-Based Data Generator for XML. In *Proceedings of WebDB*, pages 49–54, 2002.

[7] Y. Diao and M. Franklin. Query Processing for High-Volume XML Message Brokering. In *International Conference on Very Large Data Bases (VLDB)*, pages 261–272, 2003.

[8] A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *ACM SIGMOD*, pages 419–430, 2003.

[9] Z. Ives, A. Halevy, and D. Weld. An XML Query Engine for Network-Bound Data. *VLDB Journal*, 11 (4): 380–402, 2002.

[10] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *International Conference on Very Large Data Bases (VLDB)*, 2001.

[11] B. Ludascher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *International Conference on Very Large Data Bases (VLDB)*, pages 227–238, 2002.

[12] A. Marian and J. Siméon. Projecting xml documents. In *International Conference on Very Large Data Bases (VLDB)*, pages 213–224, 2003.

[13] F. Peng and S. Chawathe. XPath Queries on Streaming Data. In *ACM SIGMOD*, pages 431–442, 2003.

[14] H. Su, E. A. Rundensteiner, and M. Mani. Automaton Meets Algebra: A Hybrid Paradigm for XML Stream Processings. *Data and Knowledge Engineering (DKE) Journal*, 2006.

[15] Y. Diao, P. Fischer, M. J. Franklin, R. To. YFilter: Efficient and scalable filtering of XML documents. In *IEEE International Conference on Data Engineering (ICDE)*, pages 341–344, 2002.