

CS848 Presentation Report

Streaming XML

Yunqi (Michelle) Zhang
School of Computer Science
University of Waterloo
February, 20 , 2006

1.Introduction

XML is the preeminent data exchange format on the internet. An XML document can be seen as a rooted labeled tree, in which the children of each node are ordered. There are two ways we can query XML, namely XPath and XQuery(see introduction below). The major bottle-neck for streaming XML queries is the large amount of memory usage. In this paper, we will look at two attempt to reduce memory usage for streaming XML queries using both XQuery and XPath, and then we will also look at a space complexity lower bounds study on XPath query algorithms.

1.1 Introduction to XPath and XQuery:

XPath:

XPath is a syntax for defining parts of an XML document, it uses path expressions to navigate in XML documents.

Terminology:

Node: there are seven kinds of nodes in XPath, the most important ones that are relevant to this paper are document node, element node and attribute node. XPath treats XML as trees of nodes, with the root of the tree the 'document node/root node).

Atomic values: Atomic values are nodes with no children or parent.

Relationship of Nodes: Each element and attribute node has one parent. The element nodes may have 0+ children. Any node may have one or more siblings, ancestors and zero or more descendants.

Predicates: Predicates are used to find a specific node or a node with a specific value.

XPath uses path expressions to select nodes in an XML document. The most relevant path expressions are listed below:

<i>Expression</i>	<i>Description</i>
nodename	selects all child nodes of the node
/	selects from the root node
//	selects nodes in the document from current node no matter where we are
.	selects the current node
..	selects the parent of the current node

XQuery:

XQuery is designed to query XML data, its relationship to XML is like SQL to database tables. It is built on top of XPath. It uses path expressions to navigate through elements in an XML document, and select Nodes in SQL-like FLWOR (for, let, where, order by and return) fashion. The FLWOR expressions are the central feature of XQuery language, just as path expression is the heart of XPath language.

2. FluX, an event-based language to minimize main memory buffer usage for queries on XML documents

2.1 Introduction

Various attempts have been made with respect to the efficiency of XPath queries, for example, filtering XPath algorithm and finite state automaton approaches. However, little has been done with respect to XQueries. The most state of art XQuery engines consume main memory buffer in multiples of actual size of XML document.

Several attempts are made to optimize XQuery, includes:

1. Transducer networks – finite automaton approach. It is hard to understand comparing with other approaches and the number of states maybe exponential under certain cases.
2. Projection xml – prefiltering the data processed in xml streams to reduce size of xml query to minimize the xml document size. The problem is in real life, the need to use main-memory buffer is hard to be avoided.

To improve on this situation, a new query language, FluX is then introduced. It extends XQuery by a new construct for event-based query processing called process-stream. It optimizes the use of buffering using schema information from a DTD¹ associated with the XML.

To illustrate with an example, consider the following XQuery

```
<results>
{
  for $b in $ROOT/bib/book return
  <result> { $b/title } { $b/author } </result>
}
</results>
```

With the DTD

```
<!ELEMENT bib (book)*>
<!ELEMENT book (title|author)*>
```

¹ DTD(document type definition) - Let Σ be a set of symbols(or tag names) in a XML, the DTD is an extended context free grammar over Σ .

This query means that for each book in bibliography, output its titles/authors. From the given DTD we know that each bibliography may have more than one book associated with it and each book may have several title and several author children. As we can see, to implement this query, we can output title node inside a book as soon as it is arrived from the stream, however we will have to buffer the author children until we see the close tag for the book. Thus we can only buffer the author nodes of one book at a time for this query. However no main memory query engine currently exploits this. They would rather buffer the entire book node, or the entire title/author pair. Our FluX approach using schema constraints derived from the DTD is intended to solve this problem.

2.2 Forming FluX language from XQuery

For simplicity, we use a fragment of XQuery denoted XQuery- and demonstrate it can be transferred to FluX language. By proposition, we can generalize this result to the whole set of XQuery.

2.21 Syntax and Semantics of FluX

A simple expression is an XQuery- expression of the form α, β, γ where

- 1) α and γ are possibly empty sequences of strings and of expressions of the form “{if X then s}”, where X is a condition and s is a string.
- 2) β is either empty, “{ $\$u$ }”, or “{if X then { $\$u$ }}” for some variable $\$u$ and some condition X.
- 3) if β is of the form “{ $\$u$ }, or “{if X then { $\$u$ }}” then no atomic condition that occurs in α, β contains $\$u$.

Given the above definition, we can define the class of FluX expressions is the smallest set of expressions that are either simple or of the form:

$$s \{ \text{process-stream } \$y: \zeta \} s'$$

Where s and s' are possible empty strings, $\$y$ is a variable and ζ is a list consisting of one or more event handlers of the type:

- 1) on-first handler: on-first past(S) return α
- 2) on handler: on α as $\$x$ return Q

where $S \subseteq \text{symb}(\$y)^2$ and $\$x$ is a variable, Q is a FluX expression and α a XQuery-expression.

It is then easy to show that:

Every XQuery- query q is equivalent to the FluX query:

$$\{ \text{process-stream } \$\text{ROOT}: \text{on-fast-past}(\ast) \text{ return } q \}$$

2 let p be a regular expression, $\text{symb}(p)$ is denoted as the set of atomic symbols that occur in p

We will also define the notion of 'safe FluX queries'. A query is called safe for a given DTD if it is guaranteed that XQuery- subexpressions do not refer to paths that might still be encountered in an input stream compliant with the given DTD. It can then be shown that this notion of safety is sufficient to ensure that main memory buffers are fully populated when they are accessed by a query.

2.2.2 Translate XQuery- into FluX

The translation is done in two steps, first we transform the given XQuery- into its normal form using some normal form rewrite rules, we apply the rules in order until no further changes are possible. Next, depending on a given DTD, we rewrite this normalized query into a safe FluX query.

There are three properties for XQuery- in normal form:

- 1) All paths except the ones in conditionals are simple step paths.
- 2) Does not contain any conditional for loops.
- 3) For each subexpression of the form {if X then a} , a is either a fixed string or of the form {\$x} for some variable x.

Theorem 4.1:

The rule applications can be implemented in a way such that it terminates after $O(|Q|)$ rule applications for a XQuery- expression Q and result in a unique normalization of Q, which is equivalent to Q.

Now we can rewrite the given normalized XQuery into a FluX query. The algorithm recursively divides the given Xquery up and rewrite the expressions into FluX expressions using the given DTD constraints.

Theorem 4.3:

Given a DTD D and a normalized XQuery- Q, the rewrite algorithm runs in $O(|D|^3 + |Q|^2)$ and produces a safe FluX query that is equivalent to Q on all XML documents compliant with the given DTD.

The query engine using this algorithm can be implemented in such a way that it statically infer the buffers which are actually necessary in order to avoid superfluous buffering. The engine identifies all nodes that must be stored in buffers. We construct a prefix tree, a projection of the input document, and then optimize the prefix tree in order to restrict the amount of data being buffered.

3. XPath filtering algorithm and lower bounds complexity analysis

3.1 Introduction

Several algorithms for evaluating XPath queries over XML streams have been proposed throughout the time, including a finite automata approach. Problem with this kind of approach is that we might incur exponential blowup of the states in memory due to the loss incurred by simulating non-deterministic automata by deterministic ones. The filtering algorithm that we will introduce here avoids this problem and has a space lower bound close to minimum as we prove later.

3.2 Definitions

Forward XPath: The fragment of XPath that consists of queries that have only child, attribute or descendant axes.

Symmetric XPath: Fragment of forward XPath consisting of the queries all of whose predicates are symmetric.

Univariate XPath: Fragment of forward XPath consisting of the queries all of whose atomic predicates are univariate.

Subsumption-free XPath: The subset of queries of univariate XPath that as in addition subsumption free.

Conjunctive XPath: The subset of the mutual intersection of Symmetric XPath, Univariate XPath and Subsumption-free XPath which consists of queries that satisfies: 1) all predicates are conjunctions of atomic predicates, 2) none of their wildcard nodes and none of the children of their wildcard nodes have a descendant axis.

3.3 The algorithm

The algorithm we study here handles any query in univariate XPath. It works by processing the startElement and endElement events of the SAX interface.

On StartElement, the algorithm first checks if the document node opened matches one of the query nodes we are currently processing, if it does and the document node has not already been satisfied yet, we record the children of the query nodes that still need to be processed, and try to match these nodes.

On EndElement, checks are made towards the node that is being closed. If the node being closed is a leaf, it evaluates the predicate on that node only. After updating for the matched nodes, endElement checks if the node is the 'root', in which case we have the response of the query.

3.4 Space Complexity

In order to look at the space complexity for this algorithm and for any XPath algorithm evaluating the query on a Streaming xml documents, first we need the following definitions.

evaluation function: Any query language is associated with an evaluation function which maps query-database pairs(Q,D) into output values.

Query frontier size: The frontier of a query Q at a node u in Q is the collection of u's siblings and of its ancestors' siblings.

Recursion-depth: The recursion depth of the document is the length of the longest sequence of recursive nodes(nodes that are nested within each other and match the same query node).

Document-depth: The document depth of the document is the longest path from root to a leaf node.

Instance-optimality will be analyzed with respect to the above 3 property of queries.

Space Lower bound result:

Theorem 1. Let Q be any query in Conjunctive XPath and suppose the alphabet Σ is sufficiently large, then the space complexity of evaluation function Q on XML streams is $\Omega(FS(Q))$.

Theorem 2. Let Q be any Query in F(*), let v be the node of Q, as defined below. Then the space complexity of evaluation function of Q on XML streams is $\Omega(r)$, where r is the recursive depth of the document with respect to v.

(*) We denote by F the fragment of Conjunctive XPath that satisfies: 1) either v or an ancestor of v have a descendant axis. 2) v has at least two children with a child axis.

Theorem 3. Let Q be any query in Conjunctive XPath that has at least one node u s.t. 1) u has a child axis, 2) NTEST(u) \neq *. Then the space complexity of Q on XML streams is $\Omega(\log d)$, where d is the document depth.

The above theorems is proved to be true for any XPath algorithms. Given the above theorems,we can show that for queries in univariate XPath, the space complexity of our filtering algorithm is $O(|Q|^*r*(\log|Q| + \log d + \log r))$, the time complexity of our algorithm is $O(|D|^*|Q|^*r)$, where |Q| is the query size, |D| is the document size, r is the document recursion depth and d is the document depth. For queries in structural subsumption-free XPath and non-recursive documents, the space complexity and time complexity can be simplified to $O(FS*(\log|Q|+\log d))$ and $O(|D|^*FS)$ respectively, where FS is the query frontier size.

4. Conclusion and Future Directions

In this paper, we have showed two algorithms to optimize XQuery based and XPath based query on stream XML documents, respectively. FluX, the query language derived from XQuery provides a strong intuition for buffer-conscious query processing on structured data streams. The algorithm uses schema information to schedule FluX queries so as to reduce the use of buffers. In particular, further constraints such as cardinality constraints derived from the DTD could be used to simplify XQueries before they are rewritten into FluX, another optimization is to push if-expressions back 'up' the expression tree as soon as the other simplifications have been realized. We have also showed the minimum amount of memory requirements that any algorithm evaluating the query on a stream would need. The XPath filtering algorithm uses space close to the optimum. The future direction for this would include generalizing the result to larger classes of queries.

5. References

1. “*On the Memory Requirements of XPath Evaluation over XML Streams*” – Ziv Bar-Yossef, Marcus Fontoura, Vanja Josifovski
2. “*Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams*” - Cristoph Koch
3. “*Projecting XML Documents*” - Amelie Marian, Jerome Simeon
4. 'XPath and XQuery tutorials' - www.w3schools.com