ViteX : a Streaming XPath Processing System

Yi Chen, Susan B. Davidson and Yifeng Zheng
University of Pennsylvania, tel:(215) 573-2579 fax: (215) 898-0587
yicn@cis.upenn.edu, susan@cis.upenn.edu, yifeng@cis.upenn.edu

Abstract

We present ViteX, an XPath processing system on XML streams with polynomial time complexity. ViteX uses a polynomial-space data structure to encode an exponential number of pattern matches (in the query size) which are required to process queries correctly during a single sequential scan of XML. Then ViteX computes query solutions by probing the data structure in a lazy fashion without enumerating pattern matches.

1 Motivation

In many emerging applications such as stock market data, sports tickers, electronic personalized newspapers, and traffic information, data is presented as an XML stream. Streaming XML query evaluation has therefore attracted a lot of interest.

There are several requirements for query evaluation in these streaming environments. First, only a single sequential scan of the data is allowed. Second, it is desirable to incrementally produce and distribute query results to end users before the data is completely received. Third, since data streams can be large, a query processing algorithm must scale well in terms of processing time and space.

However, these requirements present challenges in an XML environment due to several features of query languages for XML (e.g. XPath) combined with the fact that the data is recursive:

- First, when descendant axis traversal is performed over recursive XML data, a single XML node can have multiple pattern matches to a subquery.

Consider query *Q: //section[author]//table[position]//cell* and the sample XML data in figure 1. When we process node $cell$ in line 8, we find that there are 9 ways for $cell$ to match the subquery $//section//table/cell$. Using the line number of the start tag as a subscript to distinguish XML nodes with the same tag, the pattern matches can be denoted as $(section_i, table_j, cell_8)$, where $i = 2, 3, 4$, $j = 5, 6, 7$.

- Different pattern matches to a subquery have different satisfaction on query predicates. Only a match which satisfies all predicates will contribute to the query solution. However, since data is scanned sequentially, the satisfaction of pattern matches may not be determined when the subquery pattern match is found. We therefore need to record subquery pattern matches to guarantee that correct solutions are returned.

```
1.<book>
2.  <section>
3.    <section>
4.      <section>
5.        <table>
6.          <table>
7.            <table>
8.              <cell> A </>
9.            </table>
10.          </table>
11.          <position> B </>
12.        </table>
13.      </section>
14.    </section>
15.    <author> C </>
16.  </section>
17.</book>
```

Figure 1: Sample XML data

For example, when we process line 8, we are not able to determine the predicate satisfaction of its 9 subquery pattern matches; therefore we need to record them. Later on when we process line 9, we find out that $table_7$ does not satisfy predicate $[position]$, therefore pattern matches $(section_i, table_7, cell_8)$, where $i = 2, 3, 4$, do not qualify $cell$ as a query solution and are removed. The same happens for $table_6$. Finally we find out that match $(section_2, table_5, cell_8)$ satisfies both predicates, and therefore qualifies $cell_8$ as a query solution.

These challenges are not present in a non-streaming XML query evaluation algorithm since predicates can be checked immediately by randomly accessing XML nodes. However, a streaming XML query evaluation algorithm needs to record subquery pattern matches to ensure correctness. This could be done naively by explicitly storing pattern matches, and enumerating them to test predicates. However, the number of pattern matches can be exponential, and therefore the approach has a worst case complexity which is exponential in the query size. By concisely storing pattern matches and pruning the search space without enumerating all pattern matches, ViteX (a "vite"[1] XPath processing system) achieves time and space complexity which is polynomial in the worst case for XPath [1] queries containing child axes, descendant axes, wildcards and predicates (denoted as $XP^{\{/,//,*,[]\}}$) on XML streams.

[1] "Vite" is French for "fast".

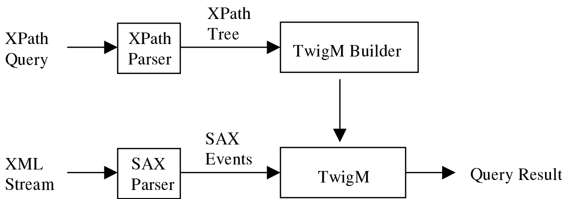

Figure 2: Architecture of the ViteX system

2 Features of the ViteX System

ViteX system has several salient features:

1. ViteX achieves polynomial time complexity in both data and query size for evaluating $XP^{\{/,//,*,[]\}}$ queries on streaming data.

2. The query processor TwigM can be constructed from an XPath query in time which is linear in the size of the query.

3. TwigM uses a compact data structure to encode pattern matches concisely rather than storing them explicitly, and therefore requires a small amount of memory. For example, experiments have shown that the memory requirement of ViteX when processing queries on a 75 MB Protein dataset [2] is stable at 1MB.

4. TwigM computes query solutions by probing the compact data structure in a lazy fashion without computing pattern matches, therefore it is very efficient.

5. ViteX not only has good theoretical time complexity but also works efficiently in practice on a variety of queries and datasets. For example, experiments with ViteX have shown that the XPath query *//ProteinEntry[reference]/@id* executing on a 75MB Protein Dataset [2] only requires 6.02 seconds (including 4.43 seconds for SAX parsing).

3 System Design

The ViteX system is composed of four modules: an XPath parser, a TwigM builder, an XML SAX parser and a TwigM machine which is the query processor, as shown in figure 2. The XPath parser takes an XPath query Q as input and generates a tree representation of the query. The TwigM builder constructs a TwigM machine according to the input query tree. The SAX parser takes an XML stream and outputs a sequence of SAX events. As SAX events stream in, TwigM changes its state according to the current state and the input event, and computes a set of XML fragments as solutions to Q.

3.1 TwigM Builder

TwigM can be built from the input query in linear time. A machine node is constructed for each query node, and they are organized in a tree structure corresponding to the query. For example, the TwigM machine for query *//section[author]//table[position]//cell* is presented in figure 3. We create a machine node for each tag and wildcard in the query. We connect the machine nodes by the edge to denote the relationship of nodes. A single-line denotes a child axis ('/'), and a double-line denotes a descendant axis ('//'). Each machine node has a stack associated with it to record its state, which is initialized to be empty. The transition functions of TwigM compute the status of the stacks according to current status and the input SAX event.

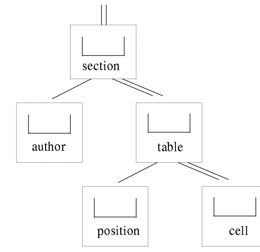

Figure 3: An example

3.2 TwigM machine

The key of TwigM is to use the stack of each machine node u to store the XML nodes that are solutions of the subquery from the root of TwigM to u. A stack node is a triplet: the *level* of the corresponding XML node, information about the *match status* of its children in the query tree, and *candidate* query solutions. On a startElement(tag, $level$) event, if tag matches the name of machine node u, and $level$ satisfies the axis denoted by u's incoming edge, we push the current XML node information to u's stack. On an endElement(tag,$level$) event, if tag matches the name of u, and $level$ equals the level of the top node in u's stack, we pop the stack. Simultaneously, we bookkeep the matching information and candidate solutions associated with the popped node to the nodes in u's parent. By keeping tracking of matching information in a recursive fashion, a node matching the root of TwigM ensures that the candidate solutions associated with it are indeed query solutions.

Recall that there is potentially an exponential number of pattern matches for an XML node which need to be recorded to evaluate queries correctly during a single sequential scan of XML. If we were to enumerate all matches, the complexity will be $O(|D|^{|Q|})$, where $|D|$ is the XML data size and $|Q|$ is the query size. By encoding the pattern matches in a compact way and probing pattern matches to compute query solutions in a lazy fashion, TwigM achieves a complexity of $O(|D||Q|(|Q|+B))$, where B is the size of candidate solutions.

References

[1] J. Clark and S. DeRose. XML Path language (XPath), November 1999. http://www.w3.org/TR/xpath.

[2] Georgetown Protein Information Resource. Protein Sequence Database, 2001. http://www.cs.washington.edu/research/xmldatasets/.