BUXT Engine in Xord:
Fragment Buffers for Streaming XSLT Transformations

Jana Dvořáková, Filip Zavoral
Charles University in Prague, Czech Rep.
dvorakova@ksi.mff.cuni.cz, zavoral@ksi.mff.cuni.cz

Abstract

We have significantly enhanced the Xord framework for streaming processing of large XML data by introducing buffers for input fragments. The analyzer examines given top-down XSLT and XSD, and generates fragments which identify parts of XSD that need to be buffered during XSLT transformation. The transformer uses auxiliary memory buffers to store temporary data according to the analyzed fragments. We describe the implementation of the engine within the Xord framework and provide evaluation tests which show that the new engine is much more memory-efficient comparing to the common XSLT processors.

1. Introduction

In this paper we focus on automatic streaming processing of XSLT transformations. Currently there is no appropriate alternative to the traditional automatic tree-based XSLT processors which are not suitable for processing of large XML data or data streams because the whole input document must be parsed and stored in a memory before the transformation.

During our previous work on the Xord project [3,4], the Xord framework for the streaming processing of XSLT transformations was designed and implemented. Each Xord engine consists of an analyzer and a transformer. The analyzer analyzes given XSLT and determines whether it can be processed by given engine. The transformers are based on a formal model called streaming XML transducer that enables us to explicitly determine the class of XSLT transformations captured and the memory consumed.

Recent SSXT engine [3] processes a subset of top-down XSLT using stack of the size proportional to the depth of the input XML document. Although the SSXT transformation algorithm is highly memory efficient, the class of possible transformations is markedly restricted - the ordering of the output nodes must follow the order of the input document.

We present a BUXT engine that overcomes these limitations. Some parts of the input document can be stored in buffers for future processing so that the output can be potentially in any order according to the input document.

The main contributions are the following:

- Design and implementation of the BUXT analyzer that statically computes schema fragments - contexts when buffering is needed based on an analysis of given schema and XSLT stylesheet,
- Design, implementation and evaluation of the BUXT transformer as an extension of SSXT by buffers and fragments.

2. Related work

Existing XQuery streaming processors (BEA/XQRL [5], FluXQuery [7], XSM [8]) are typically designed for specific purpose. Their streamability is achieved by ad-hoc optimizations and the amount of memory used for certain types of transformations is not exactly specified. XSLT automatic streaming processor SPM [6] uses known amount of memory, but it can process only very simple transformations and the class of transformations captured is not clearly characterized.

Low-level streaming languages (STX [1], StAX [10]) based on event-based programming represents another alternative for handling transformations in the streaming manner. This approach, however, requires the user to write the transformation explicitly. Other research direction deals with streaming queries [9], but this is only one of the subproblems of the whole transformation process.

3. Basic Principles

The BUXT analyzer takes two inputs: an XSLT stylesheet, and a schema. It accomplishes their static analysis and generates a set of fragments. Fragments store information on which parts of an XML document need to be buffered when the streaming transformation
is processed. The fragments are passed to the transformer.

The transformer is based on the non-buffering SSXT streaming algorithm [4]. The decision on buffering the input and when to process buffer content is taken according to the analyzed fragments.

The fragments consist of:
- `frag.tmp` - a reference to a template
- `frag.node` - a reference to an xsd node
- `frag.items` - a set of tuples:
  - `item.call` - a reference to a template call
  - `item.sob-node` - start-of-buffer xsd node
  - `item.eob-node` - end-of-buffer xsd node

A fragment identifies a subtree in the schema which require buffering when processed by the referenced template. A fragment-item identifies one of these parts, which is a subtree as well, and a specific call within the template which invokes buffering of this subtree.

The engine processes a subsets of XSLT and XSD considered in this work. We consider a top-down fragment of XSLT language. It allows matching templates with modes and top-down XPath axes. A transforming template is called by an element name and a mode. The template body consists of output elements and template calls which call other templates by an XPath expression and a mode.

A subset of XPath expression is allowed in transforming templates - they may contain child and descendant axes, and they select nodes by name.

We consider schemas without the choice constructors and recursive definitions. We represent such schema hierarchically as a schema tree. It consists of two kinds of nodes: element nodes which correspond to element types defined within schema, and constructor nodes which correspond to constructors used in the schema (sequence, *, +, ?). The relationships among element types and constructors are represented by the structure of the tree.

4. Analyzing Fragments

The analysis is driven by the tree structure of the schema tree. It searches a template which matches the current schema node and applies AnalyzeNode recursively. Following listings of pseudo-code demonstrate main ideas of described methods without nonessential technical details.

```java
AddFrag( frag);
EvalExp( el, templ,
  AnalyzeNode( calledTempl));
}

Listing 1 - AnalyzeNode
```

In case the current template does not contain any call or the current schema node is a leaf, the analysis terminates. Otherwise, the CreateFrag function is called. It finds a fragment for the current template which refers to the current schema node and adds it in case the item set is not empty. After that, the analyzing function is called recursively for all pairs such that called-temp is a template called by a current call and a schema-node is selected by the called expression in the schema tree if the evaluation starts at current schema node.

The evaluation of the expression corresponds to the evaluation against the XML tree which is formed from the schema tree by omitting all schema constructors.

```java
CreateFrag(contextNode, templ) {
  cands = CreateCands(contextNode, templ));
  foreach( i in cands) {
    foreach( j in cands) {
      if( j.call < i.call) {
        if( j.currNode > i.currNode) {
          if( j.currNode > maxNode) {
            maxNode = j.currNode;
          }
        }
      }
      if( maxNode) {
        AddFragItem( i.currNode, maxNode, i.call);
      }
    }
  }
}

Listing 2 - CreateFrag
```

The CreateFrag function first creates an empty fragment referencing to the current schema node and the template. Then particular fragment items are generated for each template call. First, the matching nodes are found for each call. The order of the matched nodes conforms to the preorder with respect to the schema tree. Then the calls and their matching nodes are processed one by one. In next steps, the algorithm determines whether a fragment item exists such that `item.sob-node = node` and `item.call = current-call`.

The item exists if and only if some end-of-buffer node is found. First, all candidates for end-of-buffer node are collected. Each such candidate must conform to the following two conditions: it must appear after the currently processed node and it must be a matching node of some call which appears before the currently processed call. This is exactly the situation when buffering is inevitable.
The maximum candidate node with respect to the preorder is chosen as the end-of-buffer node since it represents the position within the schema tree where all calls appearing before the current call have been definitely processed. In case no candidate has been found, the buffering is not needed and the fragment item is not generated.

5. Transformer Data Structures

Similarly to the original SSXT algorithm, the BUXT transformer uses a stack to remember information about particular element levels of the input document which is necessary to accomplish evaluation of XPath expressions. Two kinds of data are stored in the stack:

- DFA - sequence of current DFA states
- CC - cycle configuration

A set of DFAs is used to evaluate XPath expressions in the current template concurrently - a single DFA is associated with a single expression. Such technique has been used for example in the Y-filter algorithm [2].

The cycle configuration contains information about the currently processed part of the XSLT and the current position in the schema of the input document. The position in the schema is determined according to the current position in the input XML stream, and it is updated at each advance action. A configuration consists of the following components:

- cc.tmp - current template
- cc.call - lastly matched call
- cc.context - context schema node

During a single cycle, one template call is processed. A CC is pushed on the stack when a match is found for some expressions (i.e., a final DFA state appears in the current sequence of DFA states). Here, new cycle for processing the called template starts. A CC is popped after the called template has been processed and the control moves back to the previous template. More detailed description of SSXT stack manipulation can be found in [4].

The context schema node is a new component added first in the BUXT algorithm. It represents a position in the schema tree which corresponds to the current context tag of the transformation, i.e., the tag at which the current evaluation has started. Moreover, the transformation itself keeps a reference to the current schema node which corresponds to the currently processed tag. The pair (context schema node, current schema node) is called the current context. The current context is necessary in order to make proper decision on when to start buffering and when to process buffer contents.

The buffers are in-memory tree XML structures which are used to store temporary data for later processing. They are processed in the tree-based manner, mimicking behavior of the standard XSLT processors.

Based on the SSXT algorithm, three actions are available for stack manipulation, one action for manipulating the input XML stream and one action for generating the output XML stream:

- push DFA, push CC, pop,
- advance (advances to the next input tag),
- generate (generates output XML tags).

The BUXT algorithm uses, in addition, two more actions for manipulating buffer content:

- fill-buffer - stores the content of the current element in a buffer,
- process-buffer - buffer tree-based processing.

The decision about the buffer actions is based on the information stored in the fragments. If the current context corresponds to the fragment context and the current tag corresponds to a start-of-buffer node of one of its fragment-items then a new buffer is filled by the current subtree. Similarly, the decision about processing a buffer is made, but the end-of-buffer node is checked instead of the start-of-buffer.

6. Processing Stack Items

After an initialization, the transformer continuously calls a proper virtual processing method depending on the symbol on the top of the stack.

**Cycle configuration:** In case a new cycle starts, a sequence of initial DFA states for the current template is pushed. In case a cycle ends, last output part of the current template is generated and the previous configuration is reset.

```java
CC::Process() { 
  switch (Curr.type) { 
    case Element: 
      if (La.type == Element) 
        Push(Dfa(currTempl));
        Advance();
        break;
    case EndElement: 
      Generate(currCall);
      currTempl = templ;
      currCall = templCall;
      schemaContextNode = contextNode;
      Pop();
      break;
  }
}
```

Listing 3 - CC::Process
**DFA states.** When a start-tag is encountered, all DFAs perform a transition according to the tag name and a new sequence of states is determined. In case a final state appears in the new sequence, the transformer checks whether buffering is needed by examining all fragment items. If some of them contain a *start-of-buffer* node for the current context, a new buffer is filled with the content of the current tag. Otherwise, the content is processed in the streaming manner and a new cycle starts.

When an end-tag is encountered, the transformer first checks whether some of the buffers might be ready for processing. It again examines fragment items and selects those which contain an *end-of-buffer* node for the current context. For each such item, all associated buffers are processed in the tree-based manner. Then the streaming processing continues.

```c++
Dfa::Process() {
    switch(Curr.type) {
        case Element:
            ds = Transition( Curr.name);
            if( ! ds.HasFinalStates()) {
                if( La.type == Element)
                    Push( ds);
                    Advance();
            } else {
                calls = ds.CallsWithFinalState();
                foreach( myCall in calls) {
                    item = ProcessedLater(myCall);
                    if( item)) {
                        ReadSubtree( item);
                    } else {
                        Generate(currCall, myCall);
                        calledTempl = SelectTempl( Curr.name, myCall.mode);
                        if( calledTempl.NonCalling()) {
                            Generate();
                            currCall = myCall;
                        }
                        if( La.type == Element)
                            Push( ds);
                            Advance();
                    } else {
                        Push( CC( currTempl, myCall, schemaContextNode));
                        schemaContextNode = currSch;
                        currTempl = calledTemp;
                        currCall = 0;
                    }
            } break;
        case EndElement:
            CheckBufs();
            Advance();
            if( Curr.type == EndElement)
                Pop();
            break;
    }
}
```

Listing 4 - DFA::Process

### 7. Performance Evaluation

We have compared the BUXT algorithm space complexity against the publicly available tree-based XSLT processors (Saxon, Xalan and XsltProc) using both synthetic and real data.

![Fig. 1 - Memory requirement comparison](image1)

Fig. 1 depicts a comparison of transformation memory requirements of 10000 to 1 million nodes. All the tree-based processors consumed large amounts of memory when processing large XML data (above 100K nodes) regardless the simplicity of the transformation.

![Fig. 2 - BUXT memory consumption](image2)

The evaluation confirmed that the BUXT algorithm basically requires a memory proportional to the depth of the input XML. Fig. 2 shows a net memory
consumption of the algorithm (without libraries, runtime environment etc., in KB) processing the input data of different depth. Since the document depth is generally not depending on the document size and documents are shallow, the memory requirements for most of the XML documents are low, independent to the document size. Even for large documents like DBLP (700 MB), the BUXT algorithm required below 100 KB of net memory while the above mentioned DOM-based processors crashed or hanged after allocating about 1.5 GB of memory.

Additionally, there is an extra memory required for each fragment item detected during the transformation. The size of such memory does not depend on the whole input size but on the schema and the XSLT structure. As long as the ordering of the output document remains close to the input document (the transformation is mostly local), the space complexity remains low. The most typical example of such processing is filtering, mapping and local reordering of a huge sequence of relatively small subtrees, such as logs, structured data streams or XML databases.

On the other side, the BUXT transformer is not very suitable for some classes of transformations. The example of inappropriate transformations is swapping two large subtrees or moving a little subtree from the end of the input document to the beginning of the output. For such transformations all of the input that should be processed later must be stored into the buffers and the space complexity may achieve the tree-based processors in the worst case.

8. Conclusion

We present a BUXT engine - an enhancement of the Xord framework for efficient XSLT processing. Some parts of the input document can be stored in buffers for later processing. The analyzer can detect the context when such buffering starts and when the content of such stored buffers should be processed instead the regular input. The results of our measurements show that the engine is much less memory-consuming when processing huge data sets or data streams comparing to the common tree-based processors for a wide class of transformations.

Several issues are left for the future work. First, we intend to overcome some restrictions to XSLT and schema constructs that can be processed by the Xord engine such as conditions and choices. Next, we plan to design multipass algorithms that could be much more memory efficient for some classes of transformations at the cost of processing the input in several passes.

Acknowledgements

This work was supported in part by the National programme of research (Information society project IET100300419) and by the grant VEGA 1/3106/06.

References