The Joy of SAX

Leonidas Fegaras
University of Texas at Arlington, CSE
416 Yates Street, P.O. Box 19015
Arlington, TX 76019-19015
fegaras@cse.uta.edu

ABSTRACT

Most current XQuery implementations require that all XML data reside in memory in one form or another before they start processing the data. This is unacceptable for large XML documents. The only event-based techniques that do not require the materialization of all data in memory are based on transducers. Transducers use SAX events to evaluate XPath queries using finite state machines. SAX allows parsing and processing of XML documents in a stream fashion, but is hard to use even for simple tasks since it leaves the burden to the programmer to maintain and propagate the state between events. As is well known to people who have written non-trivial SAX event handlers, SAX can be used effectively under a disciplined use of pipes made out of SAX event handlers, where a producer object sends SAX events to a consumer object, which in turn becomes the producer for the next consumer, etc, thus forming a pipe. In this paper, I present an event-based XQuery interpreter, called *XStreamQuery*, based entirely on SAX pipelines. These pipelines consist of pipes that not only push SAX events from producers to consumers but also return feedback to the producers that allows them to cut the pipes short. This immediate feedback is difficult to achieve with transducers. Even though XStreamQuery is still a prototype system, I expect that, for simple XPath queries, it will have the same low memory overhead as transducers, although with a bit more computational overhead. But I also expect XStreamQuery to beat transducers for complex XQueries, especially for XPaths that contain predicates with embedded XQueries and for deeply-nested XQueries. More importantly, XStreamQuery allows the compositional generation of query plans, eg, one XPath step at a time, rather than requiring to look at the entire path to generate code. This makes the query plan generation very easy. It also allows path pipelines to be seamlessly integrated with the rest of XQuery. In fact, for a substantial subset of XQuery (without optimization), the plan generation code is less than 200 lines of Java while the entire XQuery processor is less than 2000 lines of Java code.

Informal Proceedings of the *First International Workshop on XQuery Implementation, Experience, and Perspectives (XIME-P)*, June 17-18, 2004, Paris, France.

1. INTRODUCTION

The reason that most current XQuery processors, such as Xalan [19], Qizx [16], and Saxon [18], fail to handle very large XML documents is that they store the entire document (in one form or another) in main memory. Some others, such as Galax [11, 5], take advantage of the structural requirements of the query to store in memory the parts needed by the query only, thus reducing the memory overhead [12]. These techniques, which resemble promoting projections before a join in relational queries, have a limited scope since they do not take advantage of content-based restrictions, which corresponds to promoting selections before a join, and are ineffective for negation and disjunction. A new evaluation framework is needed that will keep data in memory only when is absolutely necessary.

SAX [17] is an API that allows the parsing and processing of an XML document as is being read, one event at a time. In contrast to DOM, which requires the entire document to be read before it begins processing, SAX is event based: the XML parser sends events, such as the start or the end of an element, to an event handler, which in turn processes the event. The SAX API is minimal since it does not provide any means of maintaining a state between events. It leaves the burden to the programmer to do the necessary bookkeeping. Nevertheless, SAX is the only standard API that allows us to process XML data in a stream-like fashion, thus reducing the memory overhead.

Currently, there is an increased interest on using finite state machines and transducers, augmented with buffers, to process XPath queries as well as general XQueries in a stream-like fashion [9, 10, 8, 7, 14]. These approaches do an excellent job on the stream processing of simple XPath queries, but their extensions to handle predicates [8, 14] and complex XQueries [10] turned out to be very complex. I believe that the reason for this complexity is that XQuery was designed as a functional language, where an XQuery can appear at any place in a query, thus allowing the compositional translation of queries recursively, one piece at a time [3]. On the other hand, the approaches based on finite state machines require a holistic view of an XPath expression before the automaton is constructed. Furthermore, after an automaton is constructed, it needs to be incorporated into the rest of the XQuery, leaving us with only one alternative: to express the rest of the XQuery constructs with automata too, as is done in [10]. This is very hard to do even for the simplest XQuery constructs, such as sequence concatenation and element construction. Furthermore, since XQuery constructs are defined recursively, a great care must be taken to compose automata from simpler ones recursively. A notable approach that avoids the pitfalls of integrating these two incompatible models (transducers and the functional paradigm) is TypEx [15], which decomposes a query into the extraction phase (made out of automata) and computation.

The goals of this work are:

1. to build a streaming XQuery engine, called *XStreamQuery*, based entirely on SAX handlers, all the way from the points the input documents are read by the SAX parser up to the point the query results are printed;
2. this engine should consist of operators that naturally reflect the syntactic structures of XQuery and can be composed into pipelines in the same way the corresponding XQuery structures are composed to form complex queries;
3. the XQuery translation should be concise, clean, and completely compositional, so that optimizations can be easily incorporated later;
4. even though the prototype XStreamQuery system is not intended to be complete, since XQuery is a very complex language, it should be designed in such a way that all supported language features be able to be incorporated later without strain;
5. even though there is no way it can compete with the transducer-based approaches for simple XPaths, XStreamQuery should not sacrifice much on performance in terms of memory and computational overhead;
6. finally, XStreamQuery should be able to beat transducers for complex predicates and deeply nested queries.

I have to admit that, having a functional programming background, I was very reluctant to use a push-based model for query processing. A pull-based model seemed more appropriate: you pull the next event from the parser only when you need it. Moreover, how one can process two XML files at once using the push-based model? By experimenting with both approaches, I found out that it is actually easier to write handlers for push-based streams than iterators for pull-based. When you pull events from the parser you may have multiple points that try to pull the same events from the same source, such as the condition and the result paths in an XPath expression. Since the iterators that pull the events are independent, there is a racing of requests that can only be granted if events are buffered until the number of times requested reaches the number of iterators that pull the events. This complicates programming considerably. On the other hand, a push-based approach forces you to think what happens when you are provided with an event. Handling multiple documents at once is not really a problem since one can always create one thread per document reader and can let the threads push their events concurrently.

The XStreamQuery evaluation engine consists of classes that resemble SAX event handler classes. Instances of these classes, when put in sequence to form a pipeline, are stages that accept events from the previous stages and emit events to the next stages. One difference between SAX handlers and XStreamQuery handlers is that each XStreamQuery event handler returns a status that indicates whether the event passed through the pipeline successfully all the way to the terminal object of the pipeline. More specifically, the status can be *valid*, which means that the event reached the terminal object, or *invalid*, which means that the event was cut off from the pipeline at some later stage, or *unknown*, which means that the outcome of the event has not been determined yet and the event is suspended at some later stage. That way, each stage of the pipeline gets an immediate feedback from the next stages. The most common reason for an event to be cut off at some stage is that it may belong to an element not reachable by the XPath step associated with the stage. The most common reason for an event to be suspended is that a predicate may be pending (has not been found true yet) at a later stage, which causes all events to be suspended until the outcome of the predicate is known. This feedback is very important for system performance. For example, the XPath expression `//*//*/A`, when evaluated naively one step at a time without any feedback, would have been an overkill. Instead, each `//*` step gets feedback from the next stages, which allows it to cut off permutations. Most event cut-offs happen at the beginning of the pipeline, right after the SAX reader, using only one event handler, called *ShortCircuit*. ShortCircuit collects all the feedback, since the feedback is accumulated (using 3-value logic) starting from the pipeline terminals all the way back to the beginning of the pipeline. ShortCircuit propagates all SAX events until it receives an invalid status. Then, it cuts off all events up to the end of the enclosing element. Note that, returning an invalid status from an event will cause a short-circuit. Thus, only when structural constraints are not satisfied should we return an invalid status; predicates should either return a valid or unknown status, since they are based on implicit existential semantics. To my knowledge, short-circuiting the pipelines by taking into consideration the outcome of events has not been proposed before, and I believe it offers great performance benefits for long XPath queries.

Because of the XQuery FLWR expressions, a new event type had to be introduced in XStreamQuery, called endTuple, that signals the end of a tuple. Initially, when SAX events are generated by the SAX parser, only one endTuple event is emitted, immediately before the endDocument event. This indicates that the entire document is one tuple. A for-iteration in a FLWR expression though, splices one endTuple event at the end of each upper-level element. That way, during the for-iteration, each tuple will consist of one upper-level element only. At the end of the for-iteration, the endTuple events are removed. A join between two streams is done at the tuple level: each tuple of the outer stream is joined with every tuple of the inner stream (preserving document order). The XStreamQuery conditions have two modes: one used in XPaths to make decisions at the end of each upper-level element and another used in FLWR expressions to make decision at the end of each tuple.

The closest approach to ours is Joost [2, 1], which is an XSLT-like transformer built entirely with SAX event handlers. Its transformation language is STX, which is a well-behaved subset of XSL. To accommodate all possible parent XPath navigation steps, which may go all the way back to the root, Joost stores all past events in memory. Parent steps are not that important in XQuery, and most of them can be easily eliminated by simple transformations. So I decided not to support parent steps, thus limiting the expressiveness of the system, but gaining a large memory saving in return. Apache Cocoon [4] also uses the pipeline model but is based on different pipeline components. In Cocoon, an XML document is pushed through a pipeline, which begins with a generator, continues with zero or more transformers, and ends with a serializer. Even though each Cocoon transformer may accept SAX events and may generate SAX events, the transformers have been designed to perform complex tasks and are intended to be used in pipelines by programmers, not by XQuery interpreters. Another approach is the BEA/XQRL streaming XQuery processor [6], which is based on the pull-based iterator model to process SAX-like stream events (called tokens). The synchronization problem intrinsic to pull-based models was addressed by buffering tokens, which are released when the last (slowest) consumer is done. Nevertheless, their approach to XPath steps (including descendant-or-self) is similar to ours.

2. THE XStreamQuery HANDLERS

The XStreamQuery pipelines are built out of instances of subclasses of the abstract class Operator:

```
abstract class Operator {
    void suspend ();
    void release ();
    void startDocument ( int node );
    void endDocument ( int node );
    Status endTuple ( int node );
    Status startElement ( int node, String tag );
    Status endElement ( int node, String tag );
    Status characters ( int node, String text );
}
```

This class resembles a SAX Handler class but namespace URIs and element attributes are ignored for brevity. Also most methods take an extra argument node that designates the event source. More specifically, startDocument(node) is evoked at the beginning of a document; endDocument(node) is evoked at the end of a document; endTuple(node) is evoked at the end of a tuple; startElement(node,tag) is evoked at the beginning of an XML element with tagname tag; method endElement(node,tag) is evoked at the end of an XML element with tagname tag; characters(node,text) is evoked for regular character data, which is passed in text; finally, suspend()/release() suspends/releases all events up to the next endTuple event.

There are two subclasses of Operator used as terminal objects in XStreamQuery pipelines. The first is Sink(), which ignores all events:

```
class Sink extends Operator {
    boolean suspended;
    Status startElement ( int node, String tag ) {
        return (suspended) ? unknown : valid;
    }    ...
```

The second subclass, Print(), is similar to Sink(), but also prints the XML data associated with the events to the standard output. Print() is used only once for each query to print the results of the query.

Most of the other Operator classes are subclasses of the class Filter, which propagates events to the next Operator:

```
class Filter extends Operator {
    Operator next;
```

The default behavior of the Filter methods is to propagate the events to the next Operator as is. For example,

```
Status startElement ( int node, String tag ) {
   return next.startElement(node,tag);
}
```

The subclasses of Filter may choose not to propagate some of the events or may store information in their local state derived from passing events. The simplest Filter is Text that strips out all but the character events.

2.1 XPath Expressions

XPath pipelines are formed one path step at a time. For example, the XPath expression `/A//B/*/text()` is evaluated with the pipeline:

```
new Child("A",new Descendant("B",
                    new Any(new Text(...))))
```

where ... indicates the next pipeline steps.

The Child class for the XPath expression `/A` needs to keep track of the nesting level of elements (attribute nest) and whether an event should be cutoff or propagated (attribute keep):

```
Status startElement ( int node, String tag ) {
  if (nest++ == 1)
     keep = ptag.equals(tag);
  return (keep)
         ? next.startElement(node,tag)
         : invalid;
}
```

The Any class associated with the XPath step `/*` is similar to the Child class but accepts elements of any tagname.

The Descendant class for the XPath expression `//A` does not need any storage for the upper level elements with tagname “A”. For recursive XML schemas though, where an element with tagname “A” may be a descendant of another element with tagname “A”, such as a part-subpart database, a storage is necessary. The work for nested “A” elements is done at the endElement handler, where the stored events are sent to the event handler itself via a loopback. Note that, a Descendant object does not have any memory overhead if the schema is not recursive, while for nested “A” elements, the amount of storage needed is the total size of “A” elements inside each upper-level “A” element .

2.2 Predicates

XPath predicates are checked with the Predicate filter. For example, the path expression `/A[B/C="a"]/D` is evaluated with the pipeline:

```
new Child("A",
    new Predicate(true,
        new Child("B",new Child("C",new Test("a",
                                        new Sink())))),
        new Child("D",...)))
```

The first argument of Predicate indicates that this is an XPath predicate (the granularity of XPath predicates is an upper-level XML element while the granularity of a regular XQuery predicate is an entire tuple). The second argument is the condition and should be terminated with a **new** Sink(). The Test class is used for equality checking. It is similar to the Text class, but propagates the text content only if it is equal to a given string. The third Predicate argument is the continuation of the stream. The Predicate class is defined as follows:

```
class Predicate extends Filter {
    Operator condition;
    boolean pass;
    short nest;
    boolean pathp;
    Predicate ( boolean pathp,
                Operator c, Operator n ) {...}
```

The condition operator may return a valid status at any point of time. Before that time, at the beginning of each tuple, the Predicate object sends a suspend signal to the next stage, which causes the stream events to be suspended at the terminal point (the Print object). When, and if, the condition returns a valid status for the first time, the Predicate sends a release signal to the next stage and sets the attribute pass to true. This causes the immediate release of the suspended output at the terminal point and will allow the subsequent output to be immediately output. For example, in `/A[B/C="a"]/D`, if the path `B/C` appears before `D` in `/A`, then no output needs to be suspended. For example, the Predicate characters method is:

```
Status characters ( int node, String text ) {
  Status ret = condition.characters(node,text);
  if (ret.valid() && !pass) {
     pass = true;
     if (!suspended)
        next.release();
  };
  return or(next.characters(node,text),ret);
}
```

where the 'or' operator returns valid when either of its input is valid.

Note that a Predicate object does not suspend the input events locally, since this may require a larger amount of memory. Instead, it sends a suspend signal to the next stage which is propagated all the way to the printer object. That way, only the actual output to be printed is suspended, which is usually considerably smaller. Furthermore, if multiple conditions are embedded in the pipeline, the events are suspended only once, rather than multiple times. More importantly, each stage in the condition pipeline returns a feedback to the condition at event-level granularity, instead of waiting for the end of the upper-level element (for XPaths) or the end of a tuple to make the decision. This means that, for our example XPath, if there is no `B` or `B/C` path, or when there is another condition embedded inside the condition path, such as `B/C[D="b"]`, that fails, then there will be an immediate feedback that will cause the disposal of the suspended events. This immediate feedback is hard to achieve with finite-state machines.

2.3 FLWR Expressions

FLWR expressions are constructed using the For and Let classes. For example the XQuery

```
for $x in document("a.xml")/A return $x/B
```

corresponds to the pipeline:

```
new Document("a.xml",new Child("A",
    new For(100,new Select(100,new Child("B",
                new Implode(...)))))))
```

For each variable in a FLWR, such as `$x` in the above XQuery, a new node number is associated. The Let class simply propagates all events under a new node number. The For class not only propagates all events under a new node number, but also emits an endTuple event at the end of each top-level element, thus creating one tuple for each top-level element. That is, the endElement method of the For class is:

```
Status endElement ( int node, String tag ) {
   Status ret = next.endElement(fnum,tag);
   if (--nest == 0)
       ret = or(next.endTuple(fnum),ret);
   return ret;
}
```

The Select class propagates the events with a given node number only and removes the rest, thus implementing an access to a FLWR variable. The Implode filter, which is always invoked at the end of the pipeline of a FLWR expression, removes all but the last endTuple events.

2.4 Element Construction

Element construction is done using the Element and Concatenation classes. For example, `<a>{ $x/A, $y/B }</a>` is formed by first evaluating:

```
Operator v = new Concatenate(102,
                        new Element("a",...));
```

and then returning:

```
new Split(new Select(100,new Child("A",
                             new Let(102,v))),
          new Select(101,new Child("B",v)));
```

assuming that the node numbers for `$x` and `$y` are 100 and 101, respectively. For each tuple, the Concatenate operator propagates the events with a given node number before the other events. The above pipeline uses the Split class to propagate the events to two pipelines. Both pipelines end up into the Operator v, which is the Concatenate object. Thus `$x/A` and `$y/B` are mixed into one stream before they are concatenated. But since the first pipeline has been renamed to 102, for each tuple, the Concatenation propagates the events of the first pipeline before the events of the second.

Another example is:

```
<a><b>text</b><c>{ $x/C }</c></a>
```

which needs the binding:

```
Operator v = new Concatenate(101,
                  new Element("a",...));
```

and is translated into:

```
new Split(new Constant("text",new Element("b",
                             new Let(101,v))),
          new Select(100,new Child("C",
                        new Element("c",v)))))))))
```

2.5 Function Calls

Arithmetic operations, comparisons, boolean operations, as well as ordinary function calls are done with a combination of the Call and Argument classes. For example, the predicate `$x/A = $y/B` is done with the pipeline:

```
Operator v = new Call("eq",2,...);
new Split(new Select(100,new Child("A",
                         new Text(new Argument(0,v)))),
          new Select(101,new Child("B",
                         new Text(new Argument(1,v)))))
```

Both pipelines for `$x/A` and `$y/B` end up at the Operator v; the first pipeline provides the first argument while the second pipeline provides the second. For each tuple, each argument is stored in an EventQueue of the Call object and, when an endTuple event is received from each arguments, the appropriate function is performed, which in turn streams the output to the next Operator.

2.6 Joins

The only way to parse multiple documents at once in SAX is to use one thread per document and let threads emit their events concurrently. Furthermore, each Document class must behave like a cartesian product: for each endTuple event of the input stream, the entire XML document must be read and joined with the tuple. But each document can only be partitioned into tuples by some FLWR for-expression. I will describe how FLWR expressions are translated into cartesian products in the next section. Here I present an example of such a translation: The XQuery

```
for $x in document("a.xml")/a,
    $y in document("b.xml")/b
where $x/c = $y/d
return $x/e
```

needs the following bindings:

```
Operator v = new Call("eq",2,new Sink());
Operator w = new Cartesian(100,
    new Predicate(false,
          new Split(new Select(100,new Child("c",
                        new Text(new Argument(0,v)))),
                    new Select(101,new Child("d",
                        new Text(new Argument(1,v))))),
          new Child("e",new Implode(new Print())));
```

and is translated into the following program:

```
new Split(new Document("a.xml",new Child("a",
                new For(100,w))),
              new Document("b.xml",new Child("b",
                new For(101,w))))
```

The Cartesian object separates the outer stream from the inner based on the node number, 100. The inner stream is completely materialized in memory (in the form of an EventQueue). Then the outer stream is accessed one tuple at a time and each such tuple is stored in another EventQueue and is joined with each already materialized tuple of the inner stream. Since the inner and outer streams correspond to racing threads, the outer thread is blocked until the inner thread is completed. The alternative is not to store the entire inner stream in memory but to recreate the inner stream for each tuple in the outer stream. In that case, we would need to store one tuple from the inner and one tuple from the outer streams only, but we would have to pay the penalty of reading the inner document multiple times. An alternative implementation would have been using a symmetric hash join, which requires that both streams are materialized entirely in memory as hash tables, but allows asynchronous processing of documents. The use of advanced equijoin techniques, such as hash joins and sort-merge joins, complicates the XQuery translation because it requires finding the join predicates from FLWR conditions. These types of joins can only be used if there is no document order enforced to the output, such as when the result is sorted.

3. SAX PLUMBING

Compiling XQueries to XStreamQuery plans was the easiest part because these plans correspond directly to the syntactic features of XQuery. The only challenge was to form a pipe out of the XStreamQuery operators, much like the continuation passing style in functional programming languages. Figure 1 gives the rules for the translation. An XQuery e is translated into the plan $\mathcal{T}([\![e]\!], \mathbf{Print}())$, where the semantic brackets ($[\![\,]\!]$) enclose XQuery syntax (ie, they represent the abstract syntax tree associated with the syntax). Basically $\mathcal{T}([\![e]\!], c)$ translates the XQuery syntax e into a pipe of SAX Operators (the producer) that sends events to the SAX pipe c (the consumer). Function calls, $f(a_1, \ldots, a_n)$, include binary operations, such as $+$, $*$, and, $=$, $<$, etc. Element construction $< tag > \ldots < /tag >$ is equivalent to a call to element(tag, e), where e is the concatenation of the element components (using the comma operator). Function $\mathcal{F}([\![FL]\!], c)$ translates a sequence of for/let bindings FL in a FLWR expression from left to right. The second $\mathcal{F}$ rule is an improvement of the first $\mathcal{F}$ because it explores opportunities for using the Cartesian operator, which is more efficient than the implicit nested loops generated when there are more than one references to an XML document in the FLWR expression. An optimizing XQuery translator would have to explore more possibilities, including those for using symmetric hash join.

The Assign/Var operators do not correspond to any Operator class. Instead, they are used for local assignments: Assign binds a local variable to a pipeline and Var returns the value of the local variable.

As a simple example of XStreamQuery translation, the following XQuery:

```
document("data/cs.xml")/department
      /gradstudent[gpa="3.5"]/name
```

is translated into:

```
Document("data/cs.xml",ShortCircuit(
     Child(department,
           Child(gradstudent,
                 Predicate(true,
                      Child(gpa,Test("3.5",Sink())),
                      Child(name,Print()))))))
```

The following XQuery:

```
for $d in document("cs.xml")/department,
    $s in $d/gradstudent[gpa = "3.5"]
return <student>{ $d/name, $s/name }</student>
```

uses for-loops and concatenation:

```
Document("cs.xml",ShortCircuit(
   Child(department,
      For(d,
          Select(d,
             Child(gradstudent,
                Predicate(true,
                   Child(gpa,Test("3.5",Sink())),
                   For(s,
                       Assign(v0,
                         Concatenate(z,
                             Element(student,Implode(Print()))),
                         Split(Select(d,Child(name,Let(z,Var(v0)))),
                            Select(s,Child(name,Var(v0))))))))))))
```

XQueries that refer to more than one documents, such as:

```
for $x in document("a.xml")//b,
    $y in document("b.xml")/d/e
where $x/d = $y/f
return <Q>{ $x/c, $y/g }</Q>
```

use cartesian products:

```
Assign(v2,
   Cartesian(x,
       Predicate(false,
            Assign(v1,
               Call(eq,2,Sink()),
               Split(Select(x,Child(d,Text(Argument(0,Var(v1))))),
                     Select(y,Child(f,Text(Argument(1,Var(v1))))))),
            Assign(v0,
               Concatenate(z,Element(Q,Implode(Print()))),
               Split(Select(x,Child(c,Let(z,Var(v0)))),
                     Select(y,Child(g,Var(v0))))))),
   Split(Document("a.xml",ShortCircuit(Descendant(b,For(x,Var(v2))))),
         Document("b.xml",ShortCircuit(
                              Child(d,Child(e,For(y,Var(v2))))))))
```

$$
\begin{array}{rcll}
\mathcal{T}(\llbracket \text{“text”} \rrbracket, c) & = & \textbf{Constant}(\text{“text”}, c) & \\
\mathcal{T}(\llbracket \$v \rrbracket, c) & = & \textbf{Select}(v, c) & \\
\mathcal{T}(\llbracket \text{document}(url) \rrbracket, c) & = & \textbf{Document}(url, \textbf{ShortCircuit}(c)) & \\
\mathcal{T}(\llbracket \text{element}(tag, e) \rrbracket, c) & = & \mathcal{T}(\llbracket e \rrbracket, \textbf{Element}(tag, c)) & \\
\mathcal{T}(\llbracket e = \text{“text”} \rrbracket, c) & = & \mathcal{T}(\llbracket e \rrbracket, \textbf{Test}(\text{“text”}, c)) & \\
\mathcal{T}(\llbracket f(a) \rrbracket, c) & = & \mathcal{T}(\llbracket a \rrbracket, \textbf{Text}(\textbf{Argument}(0, \textbf{Call}(f, 1, c)))) & \\
\mathcal{T}(\llbracket f(a_1, \ldots, a_n) \rrbracket, c) & = & \textbf{Assign}(v, \textbf{Call}(f, n, c), \textbf{Split}(\mathcal{T}(\llbracket a_1 \rrbracket, \textbf{Text}(\textbf{Argument}(0, \textbf{Var}(v)))), & \\
 & & \quad \textbf{Split}(\ldots, \mathcal{T}(\llbracket a_n \rrbracket, \textbf{Text}(\textbf{Argument}(n-1, \textbf{Var}(v))))))) & \\
\mathcal{T}(\llbracket e_1, e_2 \rrbracket, c) & = & \textbf{Assign}(v, \textbf{Concatenate}(k, c), \textbf{Split}(\mathcal{T}(\llbracket e_1 \rrbracket, \textbf{Let}(k, \textbf{Var}(v))), \mathcal{T}(\llbracket e_2 \rrbracket, \textbf{Var}(v)))) & \\
\mathcal{T}(\llbracket /A\,path \rrbracket, c) & = & \textbf{Child}(A, \mathcal{T}(\llbracket path \rrbracket, c)) & \\
\mathcal{T}(\llbracket //A\,path \rrbracket, c) & = & \textbf{Descendant}(A, \mathcal{T}(\llbracket path \rrbracket, c)) & \\
\mathcal{T}(\llbracket /*\ path \rrbracket, c) & = & \textbf{Any}(\mathcal{T}(\llbracket path \rrbracket, c)) & \\
\mathcal{T}(\llbracket //*\ path \rrbracket, c) & = & \textbf{DescendantAny}(\mathcal{T}(\llbracket path \rrbracket, c)) & \\
\mathcal{T}(\llbracket [n]\,path \rrbracket, c) & = & \textbf{Nth}(n, \mathcal{T}(\llbracket path \rrbracket, c)) & \text{if } n \text{ is an integer} \\
\mathcal{T}(\llbracket [e]\,path \rrbracket, c) & = & \textbf{Predicate}(\text{true}, \mathcal{T}(\llbracket e \rrbracket, \textbf{Sink}()), \mathcal{T}(\llbracket path \rrbracket, c)) & \\
\mathcal{T}(\llbracket e\,path \rrbracket, c) & = & \mathcal{T}(\llbracket e \rrbracket, \mathcal{T}(\llbracket path \rrbracket, c)) & \text{otherwise} \\
\mathcal{T}(\llbracket \underline{\text{some}}\ \$v\ \underline{\text{in}}\ e_1\ \underline{\text{satisfies}}\ e_2 \rrbracket, c) & = & \mathcal{T}(\llbracket e_1 \rrbracket, \textbf{For}(v, \textbf{Predicate}(\text{false}, \mathcal{T}(\llbracket e_2 \rrbracket, \textbf{Sink}()), c))) & \\
\mathcal{T}(\llbracket FL\ \underline{\text{where}}\ e_1\ \underline{\text{return}}\ e_2 \rrbracket, c) & = & \mathcal{F}(\llbracket FL \rrbracket, \textbf{Predicate}(\text{false}, \mathcal{T}(\llbracket e_1 \rrbracket, \textbf{Sink}()), \mathcal{T}(\llbracket e_2 \rrbracket, \textbf{Implode}(c)))) & \\
\mathcal{F}(\llbracket \underline{\text{for}}\ \$v\ \underline{\text{in}}\ e\ FL \rrbracket, c) & = & \mathcal{T}(\llbracket e \rrbracket, \textbf{For}(v, \mathcal{F}(\llbracket FL \rrbracket, c))) & \text{if there is no document reference in } FL \\
\mathcal{F}(\llbracket \underline{\text{for}}\ \$v\ \underline{\text{in}}\ e\ FL \rrbracket, c) & = & \textbf{Assign}(w, \textbf{Cartesian}(v, c), \textbf{Split}(\mathcal{T}(\llbracket e \rrbracket, \textbf{For}(v, \textbf{Var}(w))), \mathcal{F}(\llbracket FL \rrbracket, \textbf{Var}(w)))) & \\
\mathcal{F}(\llbracket \underline{\text{let}}\ \$v := e\ FL \rrbracket, c) & = & \mathcal{T}(\llbracket e \rrbracket, \textbf{Let}(v, \mathcal{F}(\llbracket FL \rrbracket, c))) & \\
\mathcal{F}(\llbracket\rrbracket, c) & = & c &
\end{array}
$$

Figure 1: XStreamQuery Plumbing

4. CONCLUSION

The selling point of XStreamQuery is simplicity without a big sacrifice on performance. Its modular approach of building pipelines to evaluate XQuery scales up to any query complexity because the pipes can be connected in the same way complex queries are formed from simpler ones. To justify the performance claims made in this paper, the XStreamQuery system will be compared with a transducer-based system in the future.

The Java source of the prototype XStreamQuery system is available at `http://lambda.uta.edu/XStreamQuery.tar.gz`.

Acknowledgments: This work is supported in part by the National Science Foundation under the grant IIS-0307460.

5. REFERENCES

[1] O. Becker. Transforming XML on the Fly. XML Europe 2003, London, 5-8 May 2003

[2] O. Becker. The Joost Streaming Transformer and the Streaming Transformations for XML (STX) language. At `http://joost.sourceforge.net/`.

[3] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language. W3C Working Draft. November 2003. At `http://www.w3.org/TR/xquery/`.

[4] Apache Cocoon: An XML Web Development Framework. At `http://cocoon.apache.org/`.

[5] M. Fernandez, J. Simeon, B. Choi, A. Marian and G. Sur. Implementing XQuery 1.0: The Galax Experience. In *VLDB 2003*.

[6] D. Florescu, et al. The BEA/XQRL Streaming XQuery Processor. In *VLDB 2003*.

[7] T. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. In *ICDE'03*.

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

[9] Z. Ives, A. Levy, and D. Weld. Efficient Evaluation of Regular Path Expressions on Streaming XML Data. Technical Report, University of Washington, 2000, UW-CSE-2000-05-02.

[10] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *VLDB 2002*, pp 227–238.

[11] Galax. At `http://db.bell-labs.com/galax/`.

[12] A. Marian and J. Simeon. Projecting XML Documents. In *VLDB 2003*, pp 213-224.

[13] D. Olteanu, T. Furche, and F. Bry. An Efficient Single-Pass Query Evaluator for XML Data Streams. In *SAC 2004*, March 2004, Nicosia, Cyprus.

[14] F. Peng and S. Chawathe. XPath Queries on Streaming Data. In *SIGMOD 2003*, pp 431-442.

[15] G. Russell, M. Neumuller, and R. Connor. TypEx: A Type Based Approach to XML Stream Querying. In *WebDB 2003*.

[16] Qizx/Open. At `http://www.xfra.net/qizxopen`.

[17] SAX. At `http://www.saxproject.org/`.

[18] SAXON: The XSLT and XQuery Processor. At `http://saxon.sourceforge.net/`.

[19] Xalan. At `http://xml.apache.org/xalan-j/`.