A Fully Pipelined XQuery Processor

Leonidas Fegaras Ranjan Dash YingHui Wang
University of Texas at Arlington, Arlington, TX 76019-19015
fegaras@cse.uta.edu

ABSTRACT

We present a high-performance, pull-based streaming processor for XQuery, called XQPull, that can handle many essential features of the language, including general predicates, recursive queries, backward axis steps, and function calls, using a very small amount of caching. Our framework is based on a new type of event streams, called retarded streams, which allow multiple and nested streams to be interleaved in the same physical stream, while postponing the caching of input events until is absolutely necessary, typically at the end of the query evaluation, just before the results are ready to print.

1. INTRODUCTION

We present a high-performance, pull-based streaming processor for XQuery, called *XQPull*, that can handle many essential features of the language using a small amount of caching and having a high throughput rate and a fast latency. While other streaming approaches cache events eagerly to preserve the semantics of query operations, our system postpones caching until is absolutely necessary, typically at the end of the query evaluation, just before the results are ready to print. This is accomplished with a novel design of event streams that include events to specify the parts of the computation that need to be postponed. By postponing computations over initially large stream segments that require caching, we anticipate a later reduction of these segments by subsequent operations, thus reducing the final cache size. The postponed computations are propagated through the pipeline stages so that, after the final stream unfolding, the result is equal to that we would have gotten if we had applied the caching eagerly.

Our framework is based on the following assumptions:

- We assume that data streams are very large or even unbounded and that the nesting depth of elements is considerably smaller than the stream size. Note that, even if the entire stream can fit in memory, it would be undesirable to do so since it would result to high latency and low throughput.
- We aim at a framework that is effective for casual ad-hoc queries that produce output far smaller than the input stream. Our goal is to build a system that, for most queries, requires a memory footprint proportional to the size of the query output, rather than the query input.
- Many inefficiencies, such as some forms of backward steps, can be removed by well-known optimizations techniques. These methods can be even more effective by exploiting type information. Since XQuery is Turing complete, though, these optimizations are not complete and cannot remove all forms of inefficiency. Our framework is focused on query processing on schema-less data only, which is done after all necessary optimizations have been applied.

The contributions of our work are summarized as follows:

- We introduce a new type of event streams, called *retarded streams*, and a new framework for pull-based stream processing based on these streams. A retarded stream allows multiple and nested streams to be interleaved in the same physical stream, and postpones the caching of input events to the last stage of query evaluation.
- Each of our pipeline components uses, in the worst case, a buffer space (ie, a state) proportional to the nesting depth of the input XML stream, but not proportional to the stream size. The only possible exceptions are: the final unfolding of the retarded stream, the blocking operations (sequence concatenation, a join of two documents, set operators, and sorting), and some general existential comparisons (such as =).
- The construction of the pipeline from a query is compositional and syntax-driven. That is, each XQuery syntactic structure, such as a single XPath step, is mapped to a simple module, called an *iterator*, while a given XQuery is mapped to a pipeline by composing these iterators in the same way the corresponding XQuery syntactic structures are composed to form the query. Compositional translations result to concise, clean, and extensible implementations but require a careful design to avoid caching intermediate results.
- In contrast to related work, general predicates, recursive queries (such as $//*$), backward axis steps, and

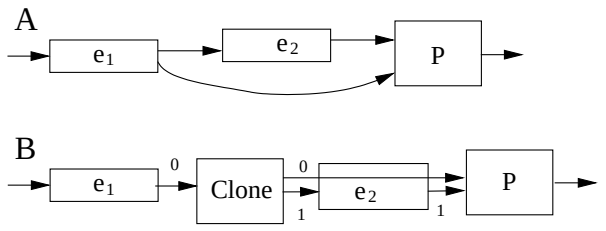

Figure 1: The push-based (A) and pull-based (B) pipelines for $e_1[e_2]$ (P is the predicate tester)

calls to user-defined functions are fully streamlined, requiring a small memory footprint.

Although our goals are similar to those of BEA [5], the contribution of our work is in the design of the retarded streams, which allows to postpone the parts of processing that require caching to the final stages of evaluation.

An alternative to pull-based, is push-based stream processing [3]. In push-based processing, the unit of a query pipeline is an event handler, which provides methods for handling the various event types. Each handler serves as both a consumer and a producer of events. As a producer, it pushes an event to the next handler in the pipeline (the consumer), by calling the appropriate consumer method (rather than pushing an object to be dispatched by the consumer). The consumer, in turn, becomes a producer and pushes events to the next handler in the pipeline, etc, until the last handler, the query printer, prints the results. Having experimented with both pull- and push-based XQuery processing [4, 3], we now believe that there are some tasks that are easier to do with a pull-based processor while some others are easier with a push-based one. For instance, a handler in a push-based pipeline can easily push the same information to multiple consumers (basically, for each event, it calls the handlers of all these consumers). This task is very important when implementing predicates, such as $e_1[e_2]$, using a compositional approach: After we construct the pipelines for e_1 and e_2, we must find a way to connect them to feed the predicate handler, P (Figure 1.A). This can be done by pushing the output of e_1 to both the input handler of the pipeline e_2 and the handler P, so that P can propagate or discard elements based on the output of e_2. Alternatively, in our pull-based framework, we had to use a special iterator, Clone, to duplicate each event in the stream so that the original event is propagated through the e_2 pipeline as is, while the replica is processed by the e_2 pipeline (Figure 1.B). While splitting streams is easier in a push-based framework, merging or joining streams is easier in a pull-based one. For example, to join two push-based streams one has to use a symmetric join, while there are numerous methods to join two pull-based streams. This is the main reason why push-based query processing is rarely used in database query processing.

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 [2, 10, 8, 7, 12, 9]. These approaches do an excellent job on the stream processing of simple XPath queries, but their extensions to handle predicates [8, 12] and complex XQueries [10] turned out to be cumbersome. While the space requirements of these approaches are, in principle, identical to those based on pull-based iterators or push-based event handlers, potentially, they can execute a little bit faster because they do not impose the overhead of method calls intrinsic to all compositional approaches. That is, while in these approaches a single stream event causes a state transition, which can be handled by a simple table lookup, in the compositional approaches, the number of method calls to handle an event is, in the worst case, proportional to the size of the pipeline. Nevertheless, by adding an acceptable amount of method call overhead, compositional approaches offer more expressive power and simplicity than holistic approaches, such those based on automata.

Finally, our retarded streams are somewhat related to lazy streams in lazy functional programming languages. That is, one can implement the query processing modules as functions from lists to lists and compose a query pipeline out of these functions. While in strict evaluation each module would construct an intermediate list to be discarded after it has been passed to the next module, in lazy evaluation, each module would implicitly process the list events through the pipeline one-event-at-a-time, without constructing the complete intermediate lists. This is how I/O streams and infinite lists are implemented in lazy functional programming languages. Alternatively, one may use program optimization techniques, such as loop fusion and partial evaluation, to fuse the list-to-list functions that form a given query pipeline to completely remove the construction of intermediate lists. Although it may have been easier to implement our system using list-to-list functions, we decided to implement our pipelines using iterators in a strict language (Java) for a finner control of laziness.

2. RETARDED STREAMS

Our retarded streams include events that correspond to those generated by the XMLPull pull-based parser [13]:

```
StartStream(stream)              // beginning of stream
EndStream(stream)                // end of stream
StartTag(stream,level,tag,attributes)
EndTag(stream,level,tag)
CData(stream,level,text)         // content text
```

but they also include an event for separating tuples generated by FLWOR loops:

```
EndTuple(stream,level)           // tuple separator
```

They may also include events that are generated by our iterators to specify that certain postprocessing is needed to generate the correct output stream, called *unfolding the stream*:

```
Suspend(stream,level)   // suspend the rest events
Release(stream,level)   // release suspended events
Discard(stream,level)   // discard suspended events
Error(stream,level,msg)// retarded error message
```

This section gives the semantics of these events. All events are associated with a stream number (since we allow multiple interleaved streams in the same retarded stream) while most events use a stream level number, explained in detail below. Briefly, if the event level is greater than zero, it indicates that this event was supposed to be cached, but instead, it was streamed immediately (ie, prematurely), with the plan to be placed at the correct position in the stream during the final stream unfolding using (hopefully less) caching. This kind of events are generated by our iterators for recursive steps, such as `//*`, without using any caching locally.

We use the notation X_l^s, Y_l^s, etc, to denote single events associated with the stream number s and the optional level l. We also use the letters α, β, etc, to denote retarded streams, which are represented as sequences of events separated by semicolon, and $\emptyset$ to denote the empty stream.

The *nesting depth* associated with a stream number s and a level l of a stream α is $\text{nest}_l^s(\alpha)$, defined as follows:

$$\begin{array}{rcll} \text{nest}_l^s(\emptyset) & = & 0 & \\ \text{nest}_l^s(\text{StartTag}_l^s;\ \alpha) & = & \text{nest}_l^s(\alpha) + 1 & \\ \text{nest}_l^s(\text{EndTag}_l^s;\ \alpha) & = & \text{nest}_l^s(\alpha) - 1 & \\ \text{nest}_l^s(X_{l'}^{s'};\ \alpha) & = & \text{nest}_l^s(\alpha) & \text{otherwise} \end{array}$$

which looks only at the events with level l and ignores the rest. We define a *top element* with stream number s and level l to be either an event CData_l^s in the stream $\alpha; \text{CData}_l^s; \gamma$, such that $\text{nest}_l^s(\alpha) = 0$, or the substream $\text{StartTag}_l^s; \beta; \text{EndTag}_l^s$ of the stream $\alpha; \text{StartTag}_l^s; \beta; \text{EndTag}_l^s; \gamma$, such that $\text{nest}_l^s(\alpha) = 0$, $\text{nest}_l^s(\beta) = 0$, and for each prefix δ of β: $\text{nest}_l^s(\delta) \neq 0$. For example, the stream

```
StartStream(0);StartTag(0,0,A,[]);CData(0,0,"x");
  EndTag(0,0,A);CData(0,0,"y");StartTag(0,0,B,[]);
    CData(0,0,"z");EndTag(0,0,B);EndStream(0);
```

has three top elements, which correspond to the XML fragments `<A>x</A>`, `y`, and `<B>z</B>`.

The Suspend/Release/Discard events are generated during the evaluation of predicates to cope with the situation in which the outcome of a predicate is determined after the events that constitute the conditional output have been received. One example is the query `/a[b/c]/d`, where the events of the `d` elements inside an `a` element arrive before the events of the `c` elements. For each Suspend event, there is a single matching Release or Discard event (in the same way StartTag/EndTag events are paired). Pairs of Suspend and their matching Release/Discard events can be nested to cope with nested predicates. The removal of these events is done at the final unfolding step in the following way:

$$\begin{array}{l} \alpha;\ \text{Suspend}_l^s;\ \beta;\ \text{Release}_l^s;\ \gamma \ \rightarrow\ \alpha;\ \beta;\ \gamma \\ \alpha;\ \text{Suspend}_l^s;\ \beta;\ \text{Discard}_l^s;\ \gamma \\ \quad \rightarrow\ \alpha;\ [\ X_{l'}^{s'} \mid X_{l'}^{s'} \in \beta,\ s' \neq s \vee l' \neq l\ ];\ \gamma \end{array}$$

That is, all the events with stream number s and level l are suspended until a matching Release or Discard is found. A Release event will release the suspended events while a Discard event will remove them.

The *predicate extent* of an XPath predicate in a stream is a top element while the predicate extent of a predicate in a FLWOR block is a tuple (ie, the events between two EndTuple events). At the beginning of a predicate evaluation, a Suspend event is generated. The first time the condition becomes true, a Release event is generated; otherwise, a Discard event is generated at the end of the predicate extent. That way, if the predicate becomes true before any output is generated, no buffering is necessary at the final stream unfolding. The only events that have to be buffered are the output events received before the predicate becomes true for the first time. The reason for the introduction of these special events is to postpone the buffering until the last stage of the pipeline (the unfolding), hoping that later stages will remove some of the suspended events. The price of laziness is that there may be unnecessary computation performed on the suspended data to be discarded at the end, which would not have been done if we had eagerly removed the unqualified events. Even worse, some of these computations may cause a run-time error. We address the latter problem by generating a special event, Error, when a run-time error occurs, to be discarded or released during unfolding.

Events with nested levels are generated during the evaluation of recursive XPath steps, such as the `//*` step and the `//part` step against recursive XML data in which parts may contain other parts recursively. For example, evaluating `//*` against the stream:

```
...;StartTag(0,0,a,[]);StartTag(0,0,b,[]);
CData(0,0,X);EndTag(0,0,b);EndTag(0,0,a);...
```

which corresponds to the XML fragment `<a><b>X</b></a>`, will result to the stream:

```
...;StartTag(0,0,a,[]);StartTag(0,0,b,[]);
StartTag(0,1,b,[]);CData(0,0,X);CData(0,1,X);
EndTag(0,0,b);EndTag(0,1,b);EndTag(0,0,a);...
```

which represents the two XML fragments `<a><b>X</b></a>` (level 0) and `<b>X</b>` (level 1), interleaved into the same stream, which can be trivially generated without buffering. That is, each stream event at depth $l > 0$ is repeated $l - 1$ times and these copies go to $0 \ldots l-2$ levels (the events at depth 0, of course, vanish).

The nested levels of a stream remain interleaved during query processing, but get unnested at the stream unfolding stage. Stream unnesting moves and promotes all stream events to level 0 so that the resulting stream is the same as the stream that we would have been generated using an eager evaluation of the recursive XPath steps. More specifically, each nesting level $l + 1$ is promoted to the nesting level l by appending the events of level $l + 1$ to the end of the current top element of level l (if exists). That is, if β is a top element of stream number s and level l in the stream $\alpha; \beta; \gamma$, then $\alpha; \beta; \gamma$ is unnested into $\alpha; \beta_1; \beta_2; \gamma$, where

$$\begin{array}{rcl} \beta_1 & = & [\ X_{l'}^{s'} \mid X_{l'}^{s'} \in \beta,\ s' \neq s \vee l' \neq l+1\ ] \\ \beta_2 & = & [\ X_l^s \mid X_{l+1}^s \in \beta\ ] \end{array}$$

That is, to remove the nesting level $l + 1$ from a stream, we promote all elements X_{l+1}^s to level l and append them to the end of the current top element of level l (if exists). Unnesting a level requires a queue of length equal to the maximum size of the top elements at that level, which could reach the stream size in the worst case. For example, the above multi-level stream is unfolded into:

```
...;StartTag(0,0,a,[]);StartTag(0,0,b,[]);
CData(0,0,X);EndTag(0,0,b);EndTag(0,0,a);
StartTag(0,0,b,[]);CData(0,0,X);EndTag(0,0,b);...
```

which corresponds to `<a><b>X</b></a><b>X</b>`. By delaying the stream unfolding up to the last moment, we anticipate the reduction/removal of events at all levels by later iterators, which may reduce the buffer requirements of the final unfolding. For example, the step `/A` in `//*/A` will work on all levels and will remove all but the `A` elements. The obvious drawback is that each iterator must now work on all stream levels, thus requiring to maintain multiple copies of its state, one for each level. The consequence of this is that the memory footprint of an iterator may now be, in the worst case, proportional to the stream nesting depth.

3. THE XQPULL ITERATORS

Our iterators are instances of subclasses of the abstract class Iterator:

```
abstract class Iterator {
    Iterator input; // the input iterator
    short stream;   // the output stream number
    void open();    // open the stream iterator
    void close();   // close the stream iterator
    Event next(); } // get the next event
```

All iterator instances take the form C($s, \ldots, input$), where s is the stream number of the output stream and *input* is the previous iterator in the pipeline. The only iterator subclass that does not have an input is Kick(s), which generates a stream with a single EndTuple:

```
StartStream(s); EndTuple(s); EndStream(s)
```

This iterator always forms the beginning of a pipeline. Parsing the input using the XMLPull pull-based parser [13] is accomplished with the iterator Document(s,s',url,input), which parses and tokenizes the document into the stream s as many times as the number of EndTuple events received from the stream s' of the input. At the beginning/end of each parsing, it emits a matching StartTag/EndTag pair to capture the implicit document root. At the end of each parsing, it emits an EndTuple event. The input iterator is typically a Kick, but in the case of an XQuery join, it may be a pipeline starting from another Document iterator, thus implementing a nested loop join.

An iterator example is Child(stream,tag,input), which implements the XPath step `/tag`. The method, next, of the Child iterator has the following code (some cases are omitted):

```
Event next () {
  while (true) {
    Event t = input.next();
    if (t.stream != stream)
      return t;
    else if (t instanceof StartTag) {
      if (nest[t.level]++ == 1) {
        pass[t.level] = tag.equals(t.tag);
        if (!pass[t.level])
          continue;
      }
    } else if (t instanceof EndTag)
      if (--nest[t.level] == 1
          && pass[t.level]) {
        pass[t.level] = false;
        return t;
      };
    if (pass[t.level])
      return t;   }}
```

For each substream l of the input stream, this method uses the counter nest[l] to keep track of the current nesting depth and the flag pass[l] to indicate whether we are currently passing through or discarding events. Since we do not know the maximum depth beforehand, these vectors are implemented as growable arrays of objects. Note that events from other streams are passed through as is.

3.1 Predicates

The pipeline of the XPath predicate $e_1[e_2]$ is formed by constructing the pipelines for e_1 and e_2 recursively, cloning the output of the pipeline e_1, and passing the cloned stream through the e_2 pipeline as is. Then, both the cloned e_1 and the output of e_2 are sent to the Predicate iterator, which embeds Suspend/Discard/Release events to the cloned stream e_1 based on the stream e_2. For example, if E is the pipeline for e, then the pipeline for e[A] can be:

```
Predicate(5,8,Child(8,A,Clone(8,5,E)))
```

Clone(cstream,stream,input) repeats each input event twice but the replica is assigned the stream number cstream. All iterators in the e_2 pipeline work only on the cstream, thus passing through stream as is. The endpoint of the predicate testing is the iterator Predicate(stream,cstream,input), which has the following method, next:

```
Event next () {
  while (true) {
    if (last != null) {
      Event e = last;
      last = null;
      return e;
    };
    Event t = input.next();
    if (t.stream == cstream) {
      if (t instanceof CData
          && !release[t.stream]) {
        release[t.stream] = true;
        return new Release(stream,t.level);
      } else continue;
    } else if (t.stream != stream)
      return t;
    if (t instanceof StartTag) {
      if (nest[t.level]++ == 0) {
        release[t.stream] = false;
        last = t;
        return new Suspend(stream,t.level);
      }
    } else if (t instanceof EndTag)
      if (--nest[t.level] == 0
          && !release[t.stream])
        last = new Discard(stream,t.level);
    return t;  }}
```

Here we use the flag release[l] to remember whether the outcome of the predicate has already been determined to be true, that is, when we have received any CData event in the stream cdata at level l that comes from the condition pipeline (e_2). For XPath predicates, the predicate extent is a top element, which means that when we encounter a StartTag at depth 0 in the original stream, we set release to false and we generate a Suspend event. When we encounter an EndTag at depth 0 and release is still false, we generate a Discard event. (Missing from the above pseudo-code is the handling of Suspend/Discard/Release events in the cstream, which requires to suspend/discard/release release[l].) For FLWOR predicates, the condition extent spans a single tuple. Thus the FLWOR Predicate operator, called FPredicate, emits Suspend/Discard events at the beginning/end of each tuple, respectively.

3.2 Recursive XPath Steps

As explained in Section 3, the iterator for the XPath step //*, called DescendantAny, generates a stream with multiple levels, one level for each nesting depth > 0. The method, next, of this iterator is the following:

```
Event next () {
  while (true) {
    if (repeat > 0)
      ... // see below
    Event t = input.next();
    if (t.stream != stream)
      return t;
    else if (t instanceof StartTag) {
      if (nest[t.level]++ > 0) {
        last = t;
        repeat = nest[t.level]-1;
      }
    } else if (t instanceof EndTag) {
      if (--nest[t.level] > 0) {
        last = t;
        repeat = nest[t.level]-1+1;
      }
    } else if (nest[t.level] > 1) {
      last = t;
```

```
        repeat = nest[t.level]-1;    }}}
```

That is, each event at nesting level i is repeated $i-1$ times. The code for repeating an event is a bit tricky:

```
if (repeat > 0) {
   Event e = last.clone();
   short i = 0;
   while (major[i] >= 0  // initially major[i]<0
          && !(major[i] == e.level
               && minor[i] == repeat))
          i++;
   if (major[i] < 0) { // new numbering
      major[i] = e.level;
      minor[i] = repeat;
   };
   e.level = i;
   repeat--;
   return e;    };
```

Since we may have nested substreams in the input stream too (which form the major level), and for each substream we need to generate multiple substreams (of minor level), we need to reassign a unique level for each different (major,minor) combination. That is, each incoming event t has a major order `t.level` and a minor order `repeat` (since `repeat` is decremented by one for each replica). The new level numbering is done with `major[i]` and `minor[i]`, which are assigned the new level i. The descendant-of step uses the same numbering scheme to handle recursive data.

3.3 FLWOR Blocks

A for-loop in a FLWOR expression is evaluated with the help of the For iterator. The For iterator removes all the EndTuple events from the input stream and inserts a new EndTuple event at the end of each top element in the input stream. That is, it generates one tuple for each top element. A let-binding, on the other hand, does not need any special iterator. Each FLWOR variable, though, (ie, a let- or a for-variable) is bound to the stream number of its domain. When a variable is used in a query, it is translated into a Clone iterator (described in Section 3.1), which duplicates the variable domain stream. (Recall that cloning duplicates each event in the stream.) When the variable is used only once in the query, it may use the original stream without cloning. At the end of the pipeline of a FLWOR block, the Implode iterator is used to remove all but the last EndTuple event from the stream. For example, the XQuery

```
for $x in fn:doc("a.xml")/A return $x/B
```

generates the pipeline:

```
Implode(2,Child(2,B,Clone(2,1,For(1,
  Child(1,A,Document(1,0,"a.xml",Kick(0)))))))
```

That is, `$x` is bound to stream 1 and is cloned to stream 2 (which can be omitted since `$x` is used once).

3.4 Constructions

Element construction is done using the Element and Concatenate classes. For example, `<Q>{$x/A,100,$x/B}</Q>` is formed by evaluating:

```
Element(3,Q,[],Concatenate(2,3,Child(3,B,
   Clone(3,0,Concatenate(1,2,Constant(2,1,"100",
      Child(1,A,Clone(1,0,input)))))))))
```

where `input` is the input pipeline and `$x` is bound to the stream number 0. The iterator Constant generates one CData event that contains the string constant for each EndTuple input event. The iterator Element embeds a StartTag at the beginning and a matching EndTag at the end of each tuple in the stream. Finally, the Concatenate iterator alternates the tuples of the two streams, starting from the first. This is accomplished by queuing the events of both streams as long as necessary to generate a single output event. During this process, the queue size may become, in the worst case, proportional to the input stream size.

3.5 User-Defined Functions

Surprisingly, it was very easy to streamline calls to user-defined functions. Obviously, we do not want to cache the input tuples and call a function one-tuple-at-a-time. Instead, we want to pass to the pipeline associated with the function body a single stream that contains multiple interleaved substreams, one substream per argument. Then this pipeline will process all the tuples of all the arguments one-event-at-a-time, without any caching (as is done for concatenation, construction, etc).

When a function f with n parameters is compiled into a pipeline, its formal parameters are bound to the stream numbers $0, \ldots, n-1$, which are reserved. Then the translation of the function body will allocate new stream numbers to iterators starting from n. The iterator pipeline of the function body is stored in a hash table, which is available at run time. This table also includes the number of parameters n, the stream number of the output, and the total number of the streams used in f. A call to f generates the iterator Call($s, f, [s_1, \ldots, s_n]$, *input*), which associates the stream numbers $s_1, \ldots, s_n$ to the parameters of f and expects the output of the call to be delivered under the stream number s. The Call iterator retrieves and clones the body of f lazily, at the first next() call, not when the iterator is opened, to accommodate recursive calls. After the f pipeline is retrieved, its starting point (ie, the Kick) is replaced by the Call input stream. All the events of the input stream of Call undergo a translation, called a precall, before they are passed through the f pipeline and a translation, called postcall, after they are generated by the f pipeline, just before they returned by Call. During precall, each input event with stream number $s_1, \ldots, s_n$ is converted to the stream number $0, \ldots, n-1$, respectively, while the other stream numbers are incremented by the total number of streams used in the f pipeline. That way, there is no overlapping of stream numbers. Of course, events of the other stream numbers are passed through the f pipeline as is, as they are supposed to, since XQuery does not support nested scopes. During the postcall stage, each event of the f pipeline that belongs to the stream number of the f output is converted to the stream number s, while the other stream numbers decrement by the total number of streams in f. This trick works for recursive functions too, with no need for a special run-time stack.

3.6 Stepping Backwards

XPath backward axis steps are notoriously difficult to implement efficiently in stream processing [1]. A decent optimizer should be able to remove most of them, but there may be some that cannot be removed. Backward axes can be implemented by cloning the stream source D immediately after is generated (such as, after the Document iterator) and propagating it through the pipeline until is used by the backward step. Then, the iterator that implements a backward axis is a special join between the incoming stream I and the cloned stream source D. This is a sliding window join

based on event timestamps (assigned to events when they are generated by a data source). Given these timestamps, we can identify all events of the cloned data source stream D that are clones of some event in I and, at the same time, to have available *all* the source events from the stream D. Here we sketch the implementation of the axis steps `ancestor::*` and `/..` (parent). Immediately before the AncestorAny iterator that implements `ancestor::*`, the cloned source D is passed through the DescendantAny iterator, which implements `//*` (Section 3.2), that is, each element of the cloned source at depth l is repeated l times. The AncestorAny iterator emits a Suspend event at the beginning of each top element of the cloned source D at any level and a matching Discard at the end of the element, except when one of its events is identical with (ie, has the same timestamp as) an event from the incoming stream I before the end of the top element, in which case it emits a Release. This method assumes that the distance between identical events from D and I does not exceed the sliding window size, which is true for most operators. Like `//*`, although it requires a fixed-size cache locally, it may require a cache proportional to the stream size during the final unfolding. This method does not work though if there is a blocking operation in the pipeline *before* the backward step that rearranges the order of events, such as sorting and concatenation. The parent axis step, works like the `/ancestor::*` step, but the synchronization in the sliding window takes into account the element depth so that only events of depth 1 in D and of depth 0 in I are under consideration.

4. XQUERY PLUMBING

Compiling an XQuery to an iterator pipeline was the easiest part of this project because these iterators correspond directly to the syntactic features of XQuery and the pipeline connects these iterators in the same way the corresponding syntactic features are put together to form the query. Figure 2 gives the rules for the translation. An XQuery e is translated into the pipeline $\mathcal{T}([\![e]\!], (\mathbf{Kick}(0), 0))$, where the semantic brackets ($[\![\,]\!]$) enclose XQuery syntax (ie, they represent the abstract syntax tree associated with the syntax). Basically, $\mathcal{T}([\![e]\!], (c, s))$ translates the XQuery syntax e into the pair (c', s'), where c' is the pipeline whose input stream comes from the output of the pipeline c and s/s' are the input/output stream numbers. Similarly, $\mathcal{F}([\![FL]\!], (c, s))$ translates the sequence of for/let bindings FL in a FLWOR block (the bottom case is $\mathcal{F}([\![\,]\!], P) = P$). The binding list σ is used for binding variables to stream numbers.

In the following examples, for better presentation, instead of using nested calls to iterator constructors, we display a query pipeline as a sequence of iterator constructions in reverse order of nesting. For example, the predicate pipeline given in Section 3.1 is displayed as:

```
1 Clone(8,5)
2 Child(8,A)
3 Predicate(5,8)
```

As a complete example, the following XQuery:

```
fn:doc("cs.xml")/department/student[gpa="3.5"]/name
```

is translated into:

```
1 Kick(0)
2 Document(1,0,"cs.xml")
3 Child(1,department)
4 Child(1,student)
5 Clone(2,1)
6 Child(2,gpa)
7 Constant(3,2,"3.5")
8 Call(2,=,[2,3])
9 Predicate(1,2)
10 Child(1,name)
```

and the following XQuery:

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

uses for-loops and concatenation:

```
1 Kick(0)
2 Document(1,0,"cs.xml")
3 Child(1,department)
4 For(1)
5 Clone(2,1)
6 Child(2,student)
7 Clone(3,2)
8 Child(3,gpa)
9 Constant(4,3,"3.5")
10 Call(3,=,[3,4])
11 Predicate(2,3)
12 For(2)
13 Clone(5,1)
14 Child(5,name)
15 Clone(6,2)
16 Child(6,name)
17 Concatenate(5,6)
18 Element(5,student,[])
19 Implode(5)
```

XQueries that use more than one document, such as:

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

use a nested loop join implicitly (the parser of `b.xml` reads the file as many times as the number of `$x` tuples):

```
1 Kick(0)
2 Document(1,0,"a.xml")
3 Descendant(1,b)
4 For(1)
5 Document(2,1,"b.xml")
6 Child(2,d)
7 Child(2,e)
8 For(2)
9 Clone(3,1)
10 Child(3,d)
11 Clone(4,2)
12 Child(4,f)
13 Call(3,eq,[3,4])
14 FPredicate(2,3)
15 Clone(5,1)
16 Child(5,c)
17 Clone(6,2)
18 Child(6,g)
19 Concatenate(5,6)
20 Element(5,Q,[])
21 Implode(5)
22 Element(5,result,[])
```

In the future, we will explore other types of joins that do not require to parse the document `b.xml` multiple times.

5. PERFORMANCE EVALUATION

Our prototype system is in its very early stage of implementation, although many essential XQuery features have already been implemented. The Java code is available at `http://lambda.uta.edu/XQPull/`. The platform used for our experiments was a 3GHz Pentium 4 processor with 1GB memory on a PC running Linux. The simulation was done using Java (J2RE 1.5.0). We used one artificially generated dataset (XMark) and one real dataset (DBLP):

Benchmark	file	size	events	time
XMark	X=doc("data.xml")	224MB	12.7	10.7
DBLP	D=doc("dblp.xml")	318MB	31.3	16.2

where events is the number of XMLPull [13] events in millions and time is the time in seconds used to tokenize the document. We used nine benchmark queries:

```
1) X//asia//item[location="Mongolia"]/quantity
2) X//item[location="Mongolia"][payment="Creditcard"]/location
3) X//*[location="Mongolia"]/quantity
4) count(X//item[location="Mongolia"]/..)
5) count(X//item[location="Mongolia"]/ancestor::asia)
6) count(X//item[location="Mongolia"]/ancestor::*//location)
```

$$
\begin{array}{rcll}
\mathcal{T}([\![\$v]\!],(c,s)) & = & (\mathbf{Clone}(s',\sigma[\$v],c),s') & \text{where } s' = \text{newStream}() \\
\mathcal{T}([\![.]\!],(c,s)) & = & (\mathbf{Clone}(s',s,c),s') & \text{where } s' = \text{newStream}() \\
\mathcal{T}([\![\text{“text”}]\!],(c,s)) & = & (\mathbf{Constant}(s',s,\text{“text”},c),s') & \text{where } s' = \text{newStream}() \\
\mathcal{T}([\![\text{fn:doc}(url)]\!],(c,s)) & = & (\mathbf{Document}(s',s,url,c),s') & \text{where } s' = \text{newStream}() \\
\mathcal{T}([\![e/A]\!],P) & = & (\mathbf{Child}(s,A,c),s) & \text{where } (c,s) = \mathcal{T}([\![e]\!],P) \\
\mathcal{T}([\![e_1[e_2]]\!],P) & = & (\mathbf{Predicate}(s_1,s_2,c_2),s_1) & \text{where } (c_1,s_1) = \mathcal{T}([\![e_1]\!],P) \\
 & & & \text{and } (c_2,s_2) = \mathcal{T}([\![e_2]\!],\mathcal{X}(c_1,s_1)) \\
\mathcal{T}([\![<tag\ attr>\{\,e\,\}</tag>]\!],P) & = & (\mathbf{Element}(s,tag,attr,c),s) & \text{where } (c,s) = \mathcal{T}([\![e]\!],P) \\
\mathcal{T}([\![e_1,\ e_2]\!],P) & = & (\mathbf{Concatenate}(s_1,s_2,c_2),s_1) & \text{where } (c_1,s_1) = \mathcal{T}([\![e_1]\!],P) \\
 & & & \text{and } (c_2,s_2) = \mathcal{T}([\![e_2]\!],(c_1,s_1)) \\
\mathcal{T}([\![f(e_1,\ldots,e_n)]\!],(c_0,s_0)) & = & (\mathbf{Call}(s_1,f,[s_1,\ldots,s_n],c_n),s_1) & \text{where } \forall i \in [1,n]:\ (c_i,s_i) = \mathcal{T}([\![e_i]\!],(c_{i-1}.s_{i-1})) \\
\mathcal{T}([\![FL\ \underline{\text{where}}\ e_1\ \underline{\text{return}}\ e_2]\!],P) & = & (\mathbf{Implode}(s_2,c_2),s_2) & \text{where } (c,s) = \mathcal{F}([\![FL]\!],P) \\
 & & & \text{and } (c_1,s_1) = \mathcal{T}([\![e_1]\!],(c,s)) \\
 & & & \text{and } (c_2,s_2) = \mathcal{T}([\![e_2]\!],(\mathbf{FPredicate}(s,s_1,c_1),s)) \\
\mathcal{F}([\![\underline{\text{for}}\ \$v\ \underline{\text{in}}\ e\ \ FL]\!],P) & = & \mathcal{F}([\![FL]\!],(\mathbf{For}(s,c),s)) & \text{where } (c,s) = \mathcal{T}([\![e]\!],P) \quad \text{and } \sigma[\$v] = s \\
\mathcal{F}([\![\underline{\text{let}}\ \$v := e\ \ FL]\!],P) & = & \mathcal{F}([\![FL]\!],(c,s)) & \text{where } (c,s) = \mathcal{T}([\![e]\!],P) \quad \text{and } \sigma[\$v] = s
\end{array}
$$

Figure 2: Building the XQuery Pipelines (only the XPath step `/A` is shown)

```
7) <result>{
      for $c in X//item
      where $c/location = "Mongolia"
      return <item>{ $c/quantity, $c/payment }</item>
   }</result>
8) D//inproceedings[author="Leonidas Fegaras"]/title
9) for $d in D//inproceedings
   where contains($d/author,"Fegaras")
   order by $d/year
   return ($d/year/text(),": ",$d/title/text(),"\n")
```

and got the following measurements:

Q	time	T	cache	CR	CL	nexts	mem
1	15	15	60	0.3%	12%	17	475
2	33	6.8	100	5%	162%	89	677
3	196	1.1	860	184%	572%	683	404
4	114	2	40	55%	473%	326	846
5	32	7	40	3%	200%	95	441
6	120	1.9	40	69%	436%	329	448
7	28	8	80	3%	168%	71	752
8	81	3.9	200	14%	154%	213	561
9	89	3.6	2391	14%	156%	194	778

where time is the execution time in seconds, T is the throughput in MB/sec, cache is the number of events cached in queues, CR is the percentage of the extra events created (ie, in addition to the input events), CL is the percentage of the created events that have been cloned, nexts is the number of next() method calls in millions, and mem is the memory used in Java (totalMemory()-freeMemory()) in KBs. We can see that the cache used in these queries is very small because the blocking operations have been postponed to the end of the evaluation, where the number of events is typically small (since it is proportional to the query output, which is expected to be small for casual queries). On the other hand, the //* and /ancestor::* steps have a considerable computational overhead.

6. CONCLUSION

From our experiments, it is now apparent that our methods can be very effective for some queries but, of course, there are cases where eagerness is better than laziness. In the future, we would like to explore more the dichotomy between these two extremes and develop an optimizer or maybe a run-time adaptive system that embeds all or parts of the stream unfolding at some well-chosen points in the pipeline that result to the best performance.

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

7. REFERENCES

[1] C. Barton, *et al.* Streaming XPath Processing with Forward and Backward Axes. In *ICDE'03*.

[2] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and Scalable Filtering of XML Documents. In *ICDE'02*.

[3] L. Fegaras. The Joy of SAX. In *XIME-P'04*.

[4] L. Fegaras. XQuery Processing with Relevance Ranking. In *XSYM'04*.

[5] D. Florescu, et al. The BEA Streaming XQuery Processor. VLDB Journal 13(3): 294-315 (2004).

[6] D. Florescu, et al. The BEA/XQRL Streaming XQuery Processor. In *VLDB'03*.

[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'03*.

[9] V. Josifovski, M. Fontoura, and A. Barta. Querying XML Streams. VLDB Journal 2004.

[10] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *VLDB'02*.

[11] Galax. At `http://www.galaxquery.org/`.

[12] F. Peng and S. Chawathe. XPath Queries on Streaming Data. In *SIGMOD'03*.

[13] XPP: An XML Pull Parser. Available at http://www.extreme.indiana.edu/xgws/xsoap/xpp/.