A Fully Pipelined XQuery Processor

Leonidas Fegaras Ranjan Dash YingHui Wang
University of Texas at Arlington
fegaras@cse.uta.edu

http://lambda.uta.edu/XQPull/

Data Stream Processing

- What is a data stream?
  - continuous, time-varying data arriving at unpredictable rates
  - continuous updates, continuous queries
  - no stored index is available
- Sought characteristics of stream processing engines
  - real-time processing
  - high throughput, low latency, fast mean response time, low jitter
  - low memory footprint
- Why bother?
  - many data are already available in stream form
    - sensor networks, network traffic monitoring, stock tickers
    - publisher-subscriber systems
    - data stream mining for fraud detection
  - data may be too volatile to index
    - continuous measurements

CSE@UTA

XML Stream Processing

- Various sources of XML streams
  - tokenized XML documents
  - sensor XML data
- Granularity
  - XML tokens (events): <tag>, </tag>, “X”, etc
  - region-encoded XML elements
  - XML fragments (hole-filler model)
- Push-based processing: SAX
  - event handlers
- Pull-based processing: XML Pull (http://www.xmlpull.org/)
  - iterator model

Our Assumptions and Goals

- Focused on very large (maybe unbounded) XML data streams
  - the nesting depth of elements is assumed to be considerably smaller than the stream size
- Aimed at casual ad-hoc XQueries that produce output far smaller than the input stream
  - *GOAL:* in the worst case, non-blocking queries may use memory proportional to the output size and the nesting depth of the input stream, but not proportional to the input stream size
- Focused on query processing on schema-less data only
  - done after all necessary optimizations have been applied

    (type information can help remove many forms of inefficiency)
- Wanted to be able to streamline *all* essential XQuery features
  - FLWOR, predicates, recursive queries, backward axes, function calls
- Striven for an efficient, concise, clean, and extensible design
- Intended to be used by lightweight clients with limited memory capacity and processing power

Background: Pipelined Processing

- It's a pull-based stream processing
  - popular in database query processing
- A pipeline is a sequence of Iterators

```
class Iterator {
    Iterator input;      // the input iterator
    void open();         // open the stream iterator
    void close();        // close the stream iterator
    Event next();        // get the next event
```

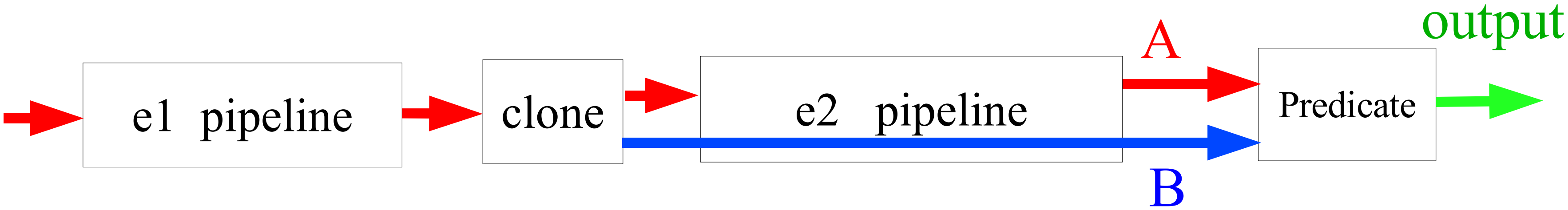
- An iterator reads events from the input stream and delivers events to the output stream
- Connected through pipelines
  - an iterator (the producer) delivers an event to the output only when requested by the next operator in pipeline (the consumer)
  - to deliver one event to the output, the producer becomes a consumer by requesting from the previous iterator as many events as necessary to produce *a single event*

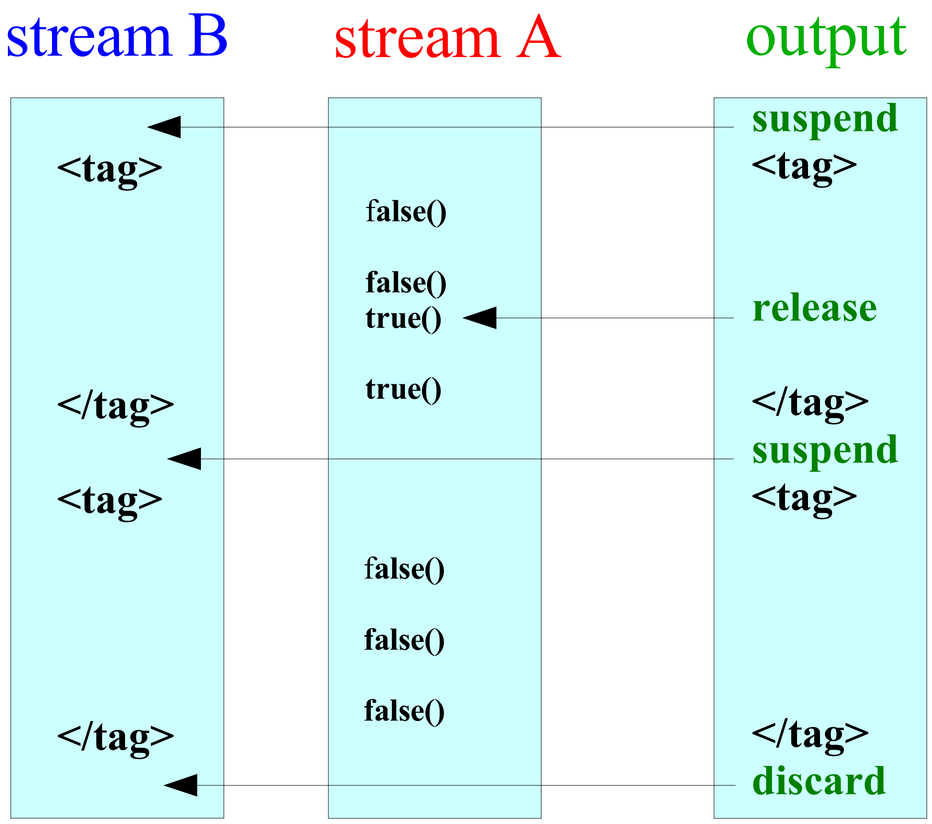
Background (cont.)

- Simple XPath steps are trivial to implement using iterators
  - not that different from transducers
- Example: the Child step (`/tag`)
  - state:
    - need a counter `nest` to keep track of the nesting depth, and
    - a flag `pass` to remember if we are currently passing through or discarding events
  - logic:
    - when we see the event `<tag>` at `nest=1`, we fall into the pass mode until we see `</tag>` at `nest=1`
    - while in pass mode, `next()` immediately returns the current event
    - while not in pass mode or `nest=0`, `next()` loops
- Hard to extend these methods to handle general predicates, recursive queries, backward steps, etc

General Predicates

- Problem: streamline `e1[e2]` without using any local cache





- Each suspend event has a matching release/discard event (like <tag> ...</tag>)
- We emit a release as soon as the predicate becomes true() at the top element
- Otherwise, we have to wait for the end of the top element to emit a discard

General Predicates (cont.)

- Simple idea: we postpone the removal of discarded events as much as possible
  - typically, to the end of query evaluation
  - ... or before a blocking operation
- Why?
  - the hope is that these segments will be reduced later by subsequent operations, thus reducing the final cache size
  - if the predicate becomes true before any output is generated from the suspended segment, no buffering is necessary
- Problems:
  - to remove the discarded events (at the end), we'd need to cache each suspended element
    - O(N) space for a stream of size N
  - each pipeline iterator must be able now to handle the new events
  - there may be unnecessary computation performed on the suspended data to be discarded later

Recursive Steps

- The XPath steps `//*` and `//part` over recursive data (ie, parts containing other parts, etc, at any depth)
- If we are strict about preserving the I/O semantics of each operator, we'd need O(N) state, for a stream size N

input

```
<a>
  <b>
    <c>
      <d>
      X
      </d>
      <d>
      Y
      </d>
    </c>
  </b>
  <b>
    <c>
      <d>
      Z
      </d>
    </c>
  </b>
</a>
```

state

state1

```
<c>
  <d>
  X
  </d>
  <d>
  Y
  </d>
</c>
```

state2

```
<c>
  <d>
  Z
  </d>
</c>
```

output

```
<b>
  <c>
    <d>
    X
    </d>
    <d>
    Y
    </d>
  </c>
</b>
<b>
  <c>
    <d>
    Z
    </d>
  </c>
</b>
```

(cont.)

```
<c>
  <d>
  X
  </d>
  <d>
  Y
  </d>
</c>
<c>
  <d>
  Z
  </d>
</c>
  <d>
  X
  </d>
  <d>
  Y
  </d>
  <d>
  Z
  </d>
```

Retarded Streams

- The reason we need a large state for //* is to append the events of depth k+1 *after* the events of depth k
- Relaxing the semantics:
  - events may appear out-of-order in a stream
    - as long as we restore the order later
- Simple idea:
  - the stream passed through the pipeline may contain multiple conceptual streams
  - each stream may include multiple levels
    - instead of deferring events by caching, we place them into a new level immediately
    - to preserve semantics, eventually, events of level k+1 must be placed after events of level k

The //* Step

- Every event of nesting depth d>0 is repeated d-1 times

input

```
<a>
  <b>
    <c>
      <d>
      X
      </d>
      <d>
      Y
      </d>
    </c>
  </b>
  <b>
    <c>
      <d>
      Z
      </d>
    </c>
  </b>
</a>
```

level0	level1	level2
		
<c>	<c>	
<d>	<d>	<d>
X	X	X
</d>	</d>	</d>
<d>	<d>	<d>
Y	Y	Y
</d>	</d>	</d>
</c>	</c>	
		
		
<c>	<c>	
<d>	<d>	<d>
Z	Z	Z
</d>	</d>	</d>
</c>	</c>	
		

- The actual physical stream is:

  <b><c><c><d><d><d>XXX</d></d></d> ...

Why Bother?

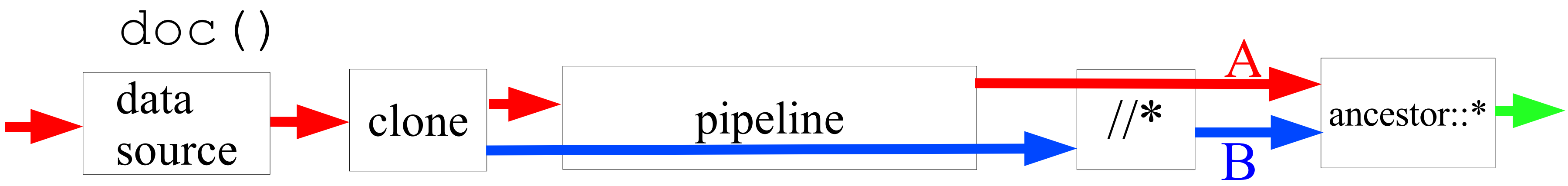
- Now recursive steps need constant size memory, but ...
- still need to move events to the right place later: O(N) again!
- ... but, hopefully, later is better than now
  - by postponing caching, we anticipate a stream reduction by subsequent operations, thus reducing the final cache size
  - works great if the query output is far smaller than the input stream
- Example: `//*/A`
  - the `//*` iterator doesn't need to know that the next step is `/A`
  - although `//*` creates many events, each event may be discarded immediately by `/A`
- The price of laziness:
  - Now each iterator must keep multiple copies of its state
    - one copy for each level
    - OK, since the maximum number of levels is the document depth

The Infamous Backward Steps

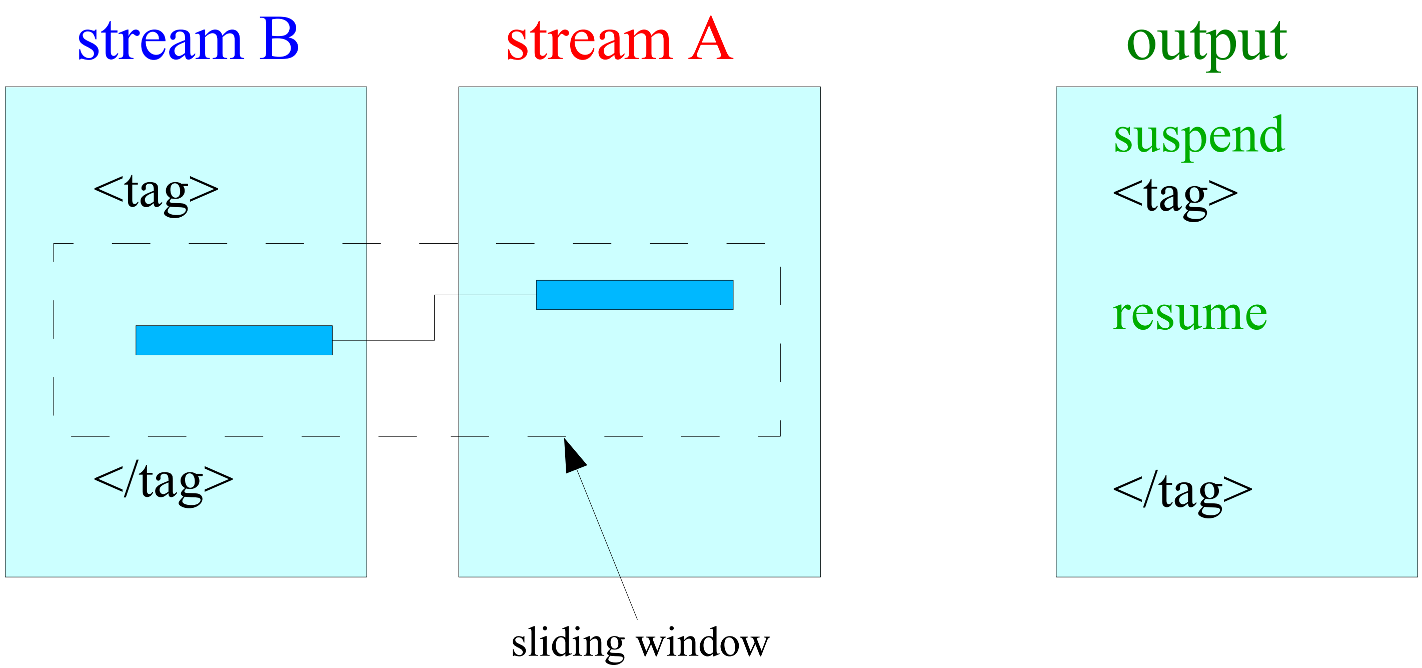
- Parent step `/..` is far more common than `ancestor::A` or `ancestor::*`
  - potentially, they may result to the whole stream
- Can we use a trick similar to `//*/A` to delay caching?
- Method:
  - clone the stream source immediately after is generated and propagate it through the pipeline until is used by the backward step
  - the iterator that implements a backward axis is a special join between the incoming stream and the cloned stream source
    - it is a sliding window semi-join that uses event timestamps to synchronize the two streams

The ancestor::* Step

- Uses the `//*` step just before the sliding window semi-join



stream A is the current context; stream B is the `doc()//*`



Backward Steps (cont.)

- Like `//*`, no caching is required locally
  - but may need O(N) at the end
- Assumes that the distance between identical events from the streams B and A does not exceed the sliding window size
  - true for most operators
  - **it does not work** if there is a blocking operation in the pipeline before the backward step that rearranges the order of events, such as sorting or concatenation

```
(for ... order by ... return ...)/ancestor::*
```

- The parent axis step (`/..`) works like the `ancestor::*` step, but the synchronization in the sliding window takes into account the element depth
  - only events of depth 1 in B and of depth 0 in A are under consideration

What About the Rest of XQuery?

- The EndTuple event separates tuples generated by FLWOR blocks
  - each inner block is driven by the outer block
    - the inner pipeline is simply appended at the end of the outer pipeline
    - an EndTuple event from the outer pipeline kicks the inner pipeline
  - let- and for-variables are bound to streams
    - a reference to a variable clones the bound stream
- A challenging query: 1
  - constants and constructions need to be kicked too
- Blocking operations
  - concatenation and sorting are straightforward
  - haven't done much about joins between documents yet
- Function calls
  - fully streamlined

Conclusion

- Did you get the feeling you've been cheated?
  - we stretched, cloned, and sliced the stream into multiple levels
  - ... but we didn't cache it!
- But, is it still stream processing?
  - yes, based on characteristics: throughput, latency, memory footprint
- Was it worthy to be so obsessed about caching?
  - promising preliminary results: up to 15 MBs/sec throughput
- Final words:
  - XQPull is still in its very early stage of implementation
  - the source code is available at http://lambda.uta.edu/XQPull/
  - please come to the demo to see it at work

To Push or to Pull? (revisited)

- Easier to implement fancy stream processing techniques using push-based processing
  - easier to split a stream: the producer sends each event to both consumers
  - our //* multilevel trick can be done by using an iterator wrapper that dispatches events based on level
- ... but, when joining two data sources, the consumer doesn't have any control of the rate the events are received from the left & right producers
  - limited choices for push-based: symmetric join
  - numerous choices for pull-based (see DBMS query processing)
- Bottom line:
  - push, if you have a single data source
  - pull, if you need to capture queries over multiple data sources and you want to use fancy join techniques