Efficient Processing of XML Update Streams

Leonidas Fegaras

University of Texas at Arlington, CSE, Arlington, TX 76019, USA
fegaras@cse.uta.edu

***Abstract*— This paper introduces a framework for processing continuous, exact queries over continuous update XML streams. Instead of eagerly performing the updates on cached portions of the stream, we propagate the updates through the query evaluation pipeline, all the way to the result display, which prints the query answers. That way, the result display prints the query results continuously, replacing old results with new. The novelty of our approach is in the use of this processing framework to unblock operations and reduce buffering by letting the operations themselves embed new updates into the stream that retroactively perform the blocking parts of the operation. Based on this framework, we present novel methods for unblocking a number of important blocking/unbounded stream operations in XQuery using a small memory footprint, such as concatenation, general predicates, descendant-or-self and backward axes, and sorting.**

I. Introduction

In the past few years, we have seen a growing interest in the development of on-line streaming applications that process stream data at high input rates under space and time constraints [1], [7]. A data stream consists of continuous, time-varying data arriving at unpredictable rates. It may be an infinite sequence of data, such as continuous measurements collected by sensors, or it may consist of finite data followed by an infinite stream of continuous updates, such as stock tickers. There are already many stream applications, such as network traffic monitoring, publisher-subscriber systems, and data stream mining, that must process stream data on-line, as they become available, to meet real-time constraints using limited resources. The frequency and the volatility of stream data make the use of standard database techniques, such as storage and indexing on disk, not suitable for long-running, continuous queries, thus necessitating the use of special data stream management systems.

Although earlier work on continuous query processing has mainly focused on relational data transmitted as streams of tuples [1], there is a recent interest in using XML as a data stream format, since it is now the language of choice for communication between co-operating systems. The XML format is more suitable for streaming complex, hierarchical data than the relational model, because the relational model enforces data normalization, which, although can be handled effectively by current relational database systems, it requires multiple streams and expensive stream joins when applied to streaming. On the other hand, although the unit of a relational stream is unquestionably a tuple, it is still an open problem to find an effective method to fragment XML data and to stream the XML fragments in such a way that it would accommodate continuous updates and would facilitate the processing of long-running, continuous queries. The most common method for XML fragmentation is XML tokenization, popularized by the SAX API for XML [15], which breaks the structure of an XML tree down into a series of linear events or tokens that can be transmitted in a stream. Many XML stream processing engines based on finite state machines [11], [9], [12] process tokenized XML documents.

The main body of earlier work on processing continuous queries over relational streams has been focused on approximation techniques that calculate approximate answers to aggregations and joins by focusing on sliding windows that contain the most recent tuples from the input streams and by using condensed synopses to summarize the state [1]. It is widely believed that without using approximation techniques, most interesting queries would be blocking (ie, they would have to wait for the end of stream to release their results) and/or unbounded (ie, their memory requirements would grow proportionally to the stream size, which may be infinite). On the other hand, most current methods for processing queries over XML streams use exact query answering techniques that are often based on finite state machines or transducers augmented with buffers [11], [9], [8], [12]. Although these techniques do an excellent job on the stream processing of simple navigational queries, their extensions to handle general predicates and other complex constructs supported by XQuery turned out to be hard.

This paper addresses the problem of processing continuous queries over streams of XML data, returning continuous, exact answers. The stream data considered are tokenized XML data with embedded updates for inserting, removing, or replacing stream subsequences that correspond to complete XML tree nodes when they are fully materialized. Our goal is to develop an architecture that can evaluate XML queries over very large XML data streams without blocking using bounded buffering.

The language of choice for querying XML data is now XQuery, which has replaced XPath as the standard query language for XML. In contrast to XPath, which supports data navigation and filtering only, XQuery allows query nesting, user-defined functions, concatenation, element construction, sorting, and joins, which are very difficult to streamline efficiently. Consider for example the following XQuery:

```
<books>{
   for $b in stream()//biblio[publisher = "Wiley"]/books
   where $b/author/lastname = "Smith"
   order by $b/price
   return <book>{ $b/title, $b/price }</book>
}</books>
```

which displays the titles and prices of all books published by Wiley and authored by Smith, sorted by their prices. Ideally, we would like to display the qualified books (ie, those books that satisfy the query conditions) in the query result display continuously, as follows: When the first qualified book is received in the stream, it is displayed immediately; the second qualified book is inserted before or after the first book, depending whether it has lower or higher price than the first, etc. In general, as soon as a qualified book is received in the stream, it is inserted in the right place in the sorted list shown in the result display. If an update comes in the input stream that updates the price of a displayed book, this book is immediately moved up or down in the sorted list in the display based on its new relative price. If the author name of a qualified book is updated to a name other than Smith, then the book is erased from the display. On the other hand, if the author of an unqualified book is updated to Smith, it is inserted into the display at the right position. More importantly, if the publisher is updated to a name other than Wiley, the entire book sequence associated with this publisher is erased from the display. The opposite happens if a publisher is changed to Wiley. None of current XQuery processors operates in this way, which is essential for a practical unblocked stream processing.

The main goal of this paper is to address blocking by relaxing the strict requirement that, at all times, the query output display should always show the correct answer up to that point. That is, we would like to optimistically display any possible output without delay and later, if necessary, to retract it or modify it somehow to make it correct. This can be accomplished by making the final output stream of the query (ie, the stream immediately before the query display) an update stream that retroactively modifies parts of the stream that have already passed through. For the previous query, every book, qualified or not, is inserted in the correct position in the display based on the book price. If a book turns out not be by Smith, then a subsequent update embedded in the output stream will erase it from the display. Later, assuming that the publisher is after the end of the book sequence, if the publisher turns out not to be Wiley, a new update will erase the entire sequence from the display. This trick would definitely unblock the query but it would display some irrelevant output, to be erased as soon as it becomes known to be irrelevant. In terms of buffering, by lazily postponing parts of a query operation all the way to the end of the query processing, we anticipate that these deferred computations would require less buffering since the query output is expected to be far smaller than the input stream. Given that updates are now parts of the data stream, we need to develop an effective framework to handle update streams. Since it is very common to have embedded updates in the input data stream too, such as continuous updates in stock feeds, we would like our framework to uniformly apply to both incoming and generated updates.

Consider for example the XPath predicate evaluation $e_1[e_2]$. This operation is binary since it combines data from two pipelines, one associated with e_1 and another with e_2. Normally, each top-level element produced by the e_1 pipeline has to be cached until the predicate becomes true, in which case it is emitted to the output, or when the end of the element is reached, in which case the element is discarded. This top-level element may span the whole stream and may become true at the end of the stream, thus making the predicate testing blocking and unbounded. More importantly, if we allow updates to the e_2 stream, then the predicate may become true at any future time, which means that we would have to cache the entire e_1 stream. For example, in //book[author="Smith"], even if a book is not authored by Smith, we would still need to cache it, because later its author may be updated to Smith. If the predicate outcome were fixed though (ie, if we knew that book authors were immutable), then this predicate could have been evaluated without caching by retroactively removing or keeping each top-level element of e_1 based on the outcome of e_2, which can be expressed as an irrevocable update. Another example, which is not blocking but requires unbounded buffering, is handling //* steps. Each qualified inner XML element must be inserted after its enclosing outer element, which requires to cache the entire inner element. This operation can be done without caching if the inner element is wrapped inside an update that specifies that its content should be inserted after the outer element.

In our framework, XML streams are sequences of SAX-like events while an XQuery stream operation is expressed as a state transformer that operates on one or more streams one-event-at-a-time, using a global state to pass information between calls. In addition, we define new types of events, not present in regular tokenized XML documents, that allow one to express well-formed updates to XML data (ie, updates that correspond to insertion, deletion, and replacement of complete XML nodes, when the stream is materialized into a tree). The main idea of this paper is to have the state transformers themselves generate new updates to unblock operations and to reduce buffering. For example, counting XML elements is a blocking operation since it must wait for the end of stream to reveal the total count. This operation is unblocked by emitting a new update to the result each time the counter is changed. This update is propagated through the pipeline until is processed by the query result display. The result display inserts, deletes, or replaces portions of the displayed text in accordance with the update. For element counting, the result display continuously displays a single number, the element counter, which is replaced every time the counter is modified. Since we assume that the query answer is far smaller than the input stream, we expect that the result display would require less buffering than in the case we had eagerly applied the updates at the point they were generated. Although there is earlier work on generating output update streams to unblock aggregations and sorting [10], [13] and on cache management for stream joins [2], our framework applies these ideas to many possible situations that cause blocking in XML stream processing.

Unfortunately, propagated updates may cause state changes to the state transformers in the pipeline. For example, when

updates associated with a predicate evaluation are passed through element counting, then the element counter that constitutes the state of the latter operation must be updated to reflect the predicate outcome. A major contribution of this work is in the development of a general framework for state adjustment that adjusts the state of a state transformer based on the incoming updates. Given a state transformer that accepts regular data streams without updates but is able to emit general streams that may contain updates to prevent blocking and to reduce buffering, our framework will automatically generate a wrapper to this state transformer that can handle any incoming update by properly adjusting its state, while applying the state transformer itself to the regular stream data. It only requires a simple state adjustment function for each state transformer that, given a past state transition, it adjusts the current state accordingly. Since any mutable region is amenable to updates, a state transformer maintains one copy of the state for each mutable region. When a new update is received by a state transformer or when its current state is adjusted because of a retroactive update that changed a past section of the stream, it causes a state transition and the new state replaces the old. By providing a general, automated method for handling incoming updates, regardless of the type of operation, we ease the tedious task of writing explicit code for each transformer that modifies its internal state to reflect the incoming updates. Finally, to reduce the total number of states cached, we have developed an efficient run-time analysis, called a *mutability analysis*, that decides whether a region is mutable or not. The state of an immutable region is removed immediately, thus keeping only the minimum required state.

The key contribution of our work is the development of a novel architecture for processing continuous queries over continuous update XML streams that has the following characteristics:

- Instead of eagerly performing the updates on cached portions of the stream, our architecture propagates the updates through the query evaluation pipeline, all the way to the result display, which prints the query answers continuously, replacing old results with new. In contrast to related methods that continuously display approximate answers by focusing on a sliding window over the stream, our framework generates exact answers continuously in the form of an update stream.
- Since the propagated updates may affect the state of the operators in the query pipeline, we provide a uniform methodology to incorporate state changes based on a simple function for state adjustment.
- The update processing framework is used in a novel way to unblock operations and reduce buffering by letting the operations themselves embed new updates into the stream that retroactively perform the blocking parts of the operation. By lazily postponing the updates as long as possible, we anticipate the reduction of the stream data through the pipeline, thus minimizing the buffering needs of these updates at the end of the pipeline.
- Based on this framework, we present novel methods for unblocking a number of important blocking/unbounded stream operations in XQuery using a small memory footprint, including concatenation, predicates, descendant-or-self and backward steps, and sorting.
- Finally, we report on a prototype implementation of an XQuery streaming engine, called XFlux, that handles many essential XQuery features and we evaluate the performance of our prototype system.

II. Simple XML Streams

In our framework, an XML stream without updates is a possibly infinite sequence of events of type $\mathcal{E}$. Each stream event $e \in \mathcal{E}$ takes one of the following forms (listed along with their abbreviations):

sS:	startStream(id)	eS:	endStream(id)
sT:	startTuple(id)	eT:	endTuple(id)
sE:	startElement(id,tag)	eE:	endElement(id,tag)
cD:	cData(id,text)		

Since we allow multiple virtual streams embedded in the same global stream, we use the number e.id to indicate the stream number of e. The events sS/eS indicate the begin/end of each virtual stream. Each virtual stream is divided into tuples generated by FLWOR loops and the events sT/eT are used to indicate the boundaries of each tuple. Finally, the events sE, eE, and cD correspond to the well-known SAX events [15] for the beginning and the end of an XML element and for a text node. For example, the XML element <name>Smith</name> is tokenized into the event sequence [sE(0,"name"), cD(0,"Smith"), eE(0,"name")].

The well-formedness, $v \in \mathcal{WF}_i$, of a sequence $v \in \mathcal{E}^*$ (ie, a list of $\mathcal{E}$ elements) for a stream number i is asserted by the following rules:

$$\forall e \in \mathcal{E} : \ e.\text{id} \neq i \ \Rightarrow \ [e] \in \mathcal{WF}_i$$
$$[\text{cD}(i,t)] \in \mathcal{WF}_i$$
$$v \in \mathcal{WF}_i \ \Rightarrow \ [\text{sE}(i,A)] +\!\!+ v +\!\!+ [\text{eE}(i,A)] \in \mathcal{WF}_i$$
$$v_1 \in \mathcal{WF}_i \wedge v_2 \in \mathcal{WF}_i \ \Rightarrow \ v_1 +\!\!+ v_2 \in \mathcal{WF}_i$$

where $+\!\!+$ is sequence concatenation. That is, the XML elements in $v \in \mathcal{WF}_i$ of stream i must be properly nested while the events of other streams are irrelevant.

An XQuery in our framework is translated into a pipeline of stages, where each stage implements a simple XQuery operation, such as an XPath step. Note that there is a single global stream that passes through the pipeline, which may consist of multiple virtual substreams. All pipeline stages, including those for binary operations, such as join, process this global stream, although they may operate on multiple substreams embedded into the global stream.

Each pipeline stage is associated with a tuple $(\mathcal{S}, s, z, i : f)$ that contains a state type $\mathcal{S}$, a state s of type $\mathcal{S}$, an initial state z of type $\mathcal{S}$, and a *state transformer* f of type $\mathcal{E} \times \mathcal{S} \rightarrow \mathcal{E}^* \times \mathcal{S}$ associated with the stream numbered i. The *effective state transformer* f' applies f to any event e with id=i, while returning the same event for all others:

$$f'(e,s) = \begin{cases} f(e,s) & \text{if } \ e.\text{id} = i \\ ([e],s) & \text{otherwise} \end{cases}$$

In general, for a stage operator that works on n substreams numbered $i_1, \ldots, i_n$, such as the binary operations join and concatenation, we have n state transformers $f_1, \ldots, f_n$, one for each stream. Then, the effective state transformer f' is:

$$f'(e,s) = \begin{cases} f_k(e,s) & \text{if } \exists k: \ e.\text{id} = i_k \\ ([e],s) & \text{otherwise} \end{cases}$$

The *sequence transformer* f^* of type $\mathcal{E}^* \times \mathcal{S} \rightarrow \mathcal{E}^* \times \mathcal{S}$ is defined recursively as follows:

$$\begin{array}{rcl} f^*([\,],s) & = & ([\,],s) \\ f^*([e_1,\ldots,e_n],s) & = & (v_1 +\!\!+ v_2, s_2) \\ & & \text{where } (v_1,s_1) = f'(e_1,s) \\ & & \text{and } (v_2,s_2) = f^*([e_2,\ldots,e_n],s_1) \end{array}$$

A sequence transformer f^* is *inert* iff

$$\forall s \in \mathcal{S}, \ \forall v \in \mathcal{WF}_i, \ \exists v' \in \mathcal{E}^* : \ f^*(v,s) = (v',s)$$

that is, if f^* does not change the state for any well-formed sequence of the input stream i (or for any i_k stream of an n-ary operation). Most XQuery operations (eg, most XPath steps) correspond to inert state transformers. Counting XML elements is an example of a non-inert transformer since its state includes a counter that increments at each XML element.

For simplicity, the function f is coded as a *state modifier* $F : \mathcal{E} \rightarrow \mathcal{E}^*$ that destructively updates the state. This is the way the state transformers are usually implemented in a procedural language. F is equivalent to f, provided that the state is cloned when necessary. For example, the state of the XPath step /tag (the tagged child-of axis) consists of two values, depth: int and pass: boolean (initially 0 and false), and is associated with the following state modifier F(e):

```
if e = sS or e = eS or e = sT or e = eT
  return [ e ]
if e = sE {
  if depth = 1 and e.tag = "tag"
    pass ← true
  depth ← depth+1
} else if e = eE {
    depth ← depth-1
    if depth = 1 and pass {
      pass ← false
      return [ e ] } }
if pass return [ e ] else return [ ]
```

The state transformer of /tag is inert because, for properly nested (ie, well-formed) XML elements, the final values of depth and pass are restored to their starting values.

The most common method for implementing the pipeline stages based on state transformers is push-based processing, in which a state transformer f is associated with a subclass F of Filter:

```
class F extends Filter {
    next: Filter
    state: S ← z
    method dispatch ( e: E ) {
        v: E*
        (v,state) ← f'(e,state)
        for each a in v do
            next.dispatch(a) } }
```

When an F object receives an event e to dispatch, it uses the current state and the effective state transformer f' to update the state and to extract new events. Then, it dispatches (pushes) the new events, one-by-one, to the next filter in the pipeline. An alternative method to push-based is pull-based processing, where an iterator reads (pulls) as many events from the input as necessary to produce a single event for the next iterator in the pipeline. The most effective stream processing method is event handling, which is basically push-based but uses a different method for each event type. That way, the producer of an event dispatches this event to the consumer (the next stage in the pipeline) by directly calling the event handler of the consumer. One example of XML event handling is SAX parsing [15]. Although our framework is independent of the streaming method used, our implementation is based on SAX.

III. XML Update Streams

An XML update stream is an XML stream extended with the following event types (along with their abbreviations):

sM:	startMutable(uid,id)	eM:	endMutable(uid,id)
sR:	startReplace(uid,id)	eR:	endReplace(uid,id)
sB:	startInsertBefore(uid,id)	eB:	endInsertBefore(uid,id)
sA:	startInsertAfter(uid,id)	eA:	endInsertAfter(uid,id)
	freeze(id)	hide(id)	show(id)

A sequence $[\text{sU}(i,j)] +\!\!+ v +\!\!+ [\text{eU}(i,j)\,]$, where sU/eU is one of the matching pairs sM/eM, sR/eR, sB/eB, and sA/eA, defines a substream sequence with number j so that $v \in \mathcal{WF}_j$, that is, the events in v with id=j are well-formed. In particular, the mutable sequence $[\text{sM}(i,j)] +\!\!+ v +\!\!+ [\text{eM}(i,j)\,]$ defines a subsequence of substream i that consists of events with number j that are amenable to updates. There are three types of updates: replace, insert before, and insert after. They can apply to either a defined mutable sequence of id=i or to an earlier update with id=i. That is, updates too are open for updates. Removing elements is done by replacing them with the empty sequence. An update id may be used multiple times for performing cascaded updates but, from all these updates with the same id, only the latest one is active and open for further updates. Freezing an id is closing it for updates (described in Section V). Update sequences can be interleaved with the stream events or with each other, provided that $\text{sR}(i,j)$ and $\text{sA}(i,j)$ come any time after the end of the region with id=i, while $\text{sB}(i,j)$ comes any time after the beginning of the region with id=i. When we hide an update with a given id, we temporarily remove the content of this update, although we leave it open for updates. We can restore the content of this update by using the show(id) event. The show/hide events are used for temporarily removing predicate output. For example, the sequence

```
[ sM(0,1), cD(1,"x"), eM(0,1), sR(1,2), cD(2,"y"), eR(1,2)
  sA(2,3), cD(3,"z"), eA(2,3), sB(1,3), cD(3,"w"), eB(1,3) ]
```

defines a mutable region with id=1, which is part the stream with id=0 and contains the text "x", and sends an update to this region (a replacement with id=2), which contains the text "y". This update basically replaces "x" with "y". Then,

the string "z" is inserted after "y" and the string "w" is inserted before "x" (which has already been replaced). After the updates are applied, the result is equivalent to the sequence [cD(0,"w"),cD(0,"y"),cD(0,"z")].

Obviously, the update events may simply be introduced at the data stream source, thus forming an update stream. Alternatively, which is the focus of this paper, the state transformers themselves may generate updates with the goal of unblocking operations and reducing buffering. That is, given a blocking or unbounded operation, we identify the parts of this operation that cause the blocking or require buffering and, instead of eagerly performing these parts, we postpone them for later by generating updates that will retroactively change the output stream that has already been generated by this operator. Consider for example the blocking operation that counts cData events at any depth. Although its state is bounded, it is blocking because it waits for the end of the stream to send the count to the next stage. We can unblock it by evaluating it using the following state modifier $F(\mathsf{e})$ with state count: int (initially 0):

```
if e = sS
  return [ e, sM(e.id,nid), cD(nid,0), eM(e.id,nid) ]
else if e = cD {
  count ← count+1
  return [ sR(nid,rid), cD(rid,count), eR(nid,rid) ]
} else return [ ]
```

where nid and rid are new ids that have not been used before. That is, $F(\mathsf{e})$ sends continuous updates on the count value, starting with 0 and sending a replacement update with the new counter value on each cD event.

IV. State Adjustment

Our update stream events can capture many kinds of stream updates. One of the goals of our work is to handle these updates gracefully with the least effort possible. More specifically, we do not want to eagerly apply the updates during processing, because these updates may be blocking or may require unbounded state. Instead, we want to propagate these updates through the processing pipeline, all the way to the result display, which prints the query answers. But we cannot just simply propagate the updates, because these updates may actually change the states of state transformers. For example, if a state transformer counts elements and we insert a new element, then this counter should be incremented to reflect this update. We can incorporate state changes to those state transformers that are not inert by adjusting their states after each update. In our framework, this is done with the help of a function adjust(s_1, s_2, s_3) of type $\mathcal{S} \times \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ that must be defined for each non-inert state transformer. It must satisfy the following properties $\forall s_1, s_2, s_3 \in \mathcal{S}$ and $\forall v \in \mathcal{WF}_i$:

$$\begin{aligned} \text{adjust}(s_1, s_2, s_2) &= s_1 \\ \text{adjust}(s_1, s_1, s_2) &= s_2 \\ \text{adjust}(f^*(v, s_1), s_2, s_3) &= f^*(v, \text{adjust}(s_1, s_2, s_3)) \end{aligned}$$

The function call, adjust(s_1, s_2, s_3), given that the state s_2 has been changed to s_3 by some earlier update, it adjusts the state s_1 to reflect this update. In particular, from the last two properties, we have:

$$\text{adjust}(f^*(v, s_1), s_1, s_2) = f^*(v, s_2)$$

which indicates that, if s_1 were changed to s_2, then $f^*(v, s_1)$ is adjusted to $f^*(v, s_2)$, as expected. For an inert state transformer, $\forall s_1, s_2, s_3 :$ adjust$(s_1, s_2, s_3) = s_1$, that is, previous state transitions do not affect the state s_1.

For simplicity, the adjust function of a state transformer is coded as the method Adjust(s_1, s_2) of type $\mathcal{S} \times \mathcal{S} \to \mathcal{E}^*$ that, when applied to the state s of the transformer, it destructively modifies the state using $s \leftarrow$ adjust(s, s_1, s_2) and returns an optional list of events that are embedded in the stream at the time of adjustment. The ability to embed new events makes state adjustment more powerful without requiring any extension to semantics. For example, the state transformer that counts cData at any depth (given in Section III) needs the following adjustment:

count ← count + (s_2.count − s_1.count); return [ ]

That is, if s_1.count changes to s_2.count due to some update, then the value of count after any subsequent update should be adjusted as shown.

In our framework, all non-inert state transformers used in our XQuery implementation are adjustable, except a very important one: the final display of the query result. For this component, we had to provide explicit code to handle every kind of event, including the update events, which cause the removal or insertion of text to the display.

Our goal is, for each adjustable state transformer, to provide a fixed wrapper that propagates the updates and correctly adjusts the state based on the adjust function exclusively. Each region open to updates (such as a mutable region or a region that has already been updated) is associated with starting and ending states. A new update, not only has to calculate its own starting/ending states, but should also adjust all those states associated with subsequent regions in the stream. The state adjustments are applied at the end of the update only. Given that for each new update, only the subsequent regions that are open to updates need to have theirs states adjusted, we assign a timestamp, order[id], to each region that reflects the relative order of this region in the stream, if these updates had been applied eagerly.

The *wrapper state transformer*, $\mathcal{W}$, handles the update events using the effective state transformer f' and the state adjustment exclusively. The state of $\mathcal{W}$ contains the mappings:

start:	id → $\mathcal{S}$	the starting state
end:	id → $\mathcal{S}$	the current/ending state
shadow:	id → $\mathcal{S}$	a copy of the ending state

The body of $\mathcal{W}(e)$ is:

```
if e is a regular stream event {
  (v, end[e.id]) ← f'(e,end[e.id])
  return v
} else { update the states (described below); return [ ] }
```

That is, for non-update events, the wrapper state transformer works directly on the current state, which is end[e.id]. When an update event, U(id,uid), is received, the states are updated as follows, based on the type of U:

```
sM, sA:    start[uid] ← end[id]; end[uid] ← end[id]
sR, sB:    start[uid] ← start[id]; end[uid] ← start[id]
eM:        end[id] ← end[uid]
eR:        adj(uid,end[id],end[uid])
eA, eB:    adj(uid,start[uid],end[uid])
```

where adj(uid,s_1,s_2) adjusts all subsequent update regions:

```
for each i ≠ uid : order[i] > order[uid] do
    start[i] ← adjust(start[i],s1,s2)
    end[i] ← adjust(end[i],s1,s2)
    shadow[i] ← adjust(shadow[i],s1,s2)
```

where order[id] is the timestamp of an update id, which is derived incrementally, when the update is received, as shown below. The show/hide events use the shadow state:

```
hide(uid):  adj(uid,end[uid],start[uid]);
            shadow[uid]← end[uid]; end[uid]← start[uid]
show(uid):  adj(uid,end[uid],shadow[uid]);
            end[uid]← shadow[uid]
```

The relative order of an update is set at the beginning of each update:

```
sM(i,j): order[j] ← order[i]      sR(i,j): order[j] ← order[i]
sA(i,j): order[j] ← (order[i] + min{order[k] |
                                   order[k] > order[i] })/2
sB(i,j): order[j] ← (order[i] + max{order[k] |
                                   order[k] < order[i] })/2
```

where the order[i] of sS(stream,i) is 1 and the min/max default values are 0/1.

V. Reducing Buffering

In our framework, every update passing through a state transformer is associated with a number of copies of the state, which may add up to a large number of states. Potentially, the data stream emitted at the data source may have every single XML element open for updates, and thus each state transformer would have to create as many state instances as the number of incoming XML nodes. Fortunately, most update streams need to have only a small number of nodes open for updates. For example, in a stock ticker stream, stock names are typically immutable, while stock quotations are mutable. Furthermore, we would like the stream consumer to be able to choose which updates to accept and which ones to ignore. Ignoring updates over an update region is the same as making the region immutable.

This information about immutable regions is used by our framework to reduce buffering. Consider, for example, the query that retrieves the IBM stock quotation. We would expect this query to print the current IBM quotation, and, when there is an update, to replace the old with the new quotation in the answer display. But if the stock name were mutable, then our system would need an unbounded state to store information about every quotation, regardless of name, because, potentially, every stock name may be changed to IBM in the future. Therefore, if we are not careful, any predicate would always require unbounded state. To address this problem, we classify update ids into fixed (closed to updates) and not fixed (open to updates), and we use the global mapping, fix: id → boolean, to indicate this fact. A mutable region in the incoming data stream emitted at the data source is considered not fixed and can be updated, except when the user indicates that is not interested in these updates, in which case this region is classified as fixed. At each update event sU(id,uid) (ie, the beginning of any update), regardless if it is incoming or generated, we copy fix[id] to fix[uid]. Then, during any state adjustment, only the ids that are not fixed are adjusted. In addition, when a state transformer sees that a fix[id] is true, it removes the states for id.

Related to generated updates, it is often possible to decide when to close an update id to future updates, because we often know exactly the scope of a generated update. The special event freeze(id) sets fix[id] to true. For example, for the query that retrieves the IBM stock quotation, we embed the quotation of every stock name in an update that is shown or hidden based on whether the stock name is IBM or not. By using the show/hide events to show/hide quotations, we can withdraw this decision when the stock name changes. But if the stock name were fixed (ie, if were immutable or inside an update region with a fixed id), then instead of using the show/hide events, we could keep or remove the quotation, which is an irrevocable decision but does not require any buffering. This irrevocable update can be generated by the predicate transformer by freezing the update id (as explained in Section VI-B).

VI. Unblocking Operations

In this section, we are focusing on some important XQuery operations that are blocking and/or require unbounded state and we are presenting algorithms to alleviate these problems based on update streams.

A. Concatenation

The concatenation of two sequences (represented as sub-streams in our framework) is a blocking operation that requires unbounded buffering (the worst case is when the left stream consists of one event coming after the right stream, but must be moved before the right stream). To make its state bounded, each tuple received from the right stream is wrapped by a mutable region and each tuple received from the left stream is wrapped by an insert-before update to retroactively move it before the left tuple. Since concatenation is binary, it requires two state modifiers, one, $F_1(e)$, for the left stream that removes the sT/eT events but leaves the rest unchanged:

```
if e = sT or e = eT return [ ] else return [ e ]
```

and another $F_2(e)$, for the right stream:

```
if e = sT
  return [ sT(nid), sM(nid,right), sB(right,left) ]
else if e = eT
  return [ eB(right,left), eM(nid,right), eT(nid) ]
else return [ e ]
```

where left/right are the stream ids of the left/right input streams and nid is a new update id that has not been used before. For example, to append stream 0 after stream 1 in:

```
[ sT(0), sT(1), cD(0,"x"), cD(1,"y"),
  cD(0,"z"), cD(1,"w"), eT(0), eT(1) ]
```

stream 1 is converted to an insert-before update:

```
[ sT(2), sM(2,1), sB(1,0), cD(0,"x"), cD(1,"y"),
  cD(0,"z"), cD(1,"w"), eB(1,0), eM(2,1), eT(2) ]
```

B. General Predicates

A general XPath predicate of the form $e_1[e_2]$ in XQuery, where e_1 and e_2 are arbitrary XQuery expressions, has the following semantics: When e_1 is evaluated, it returns a sequence of XML elements. The output of $e_1[e_2]$ consists of all those elements from this sequence that satisfy the predicate e_2. The pipeline of $e_1[e_2]$ is formed by putting in sequence the pipeline of e_1, followed by the pipeline of e_2 (provided that they generate substreams of a different stream number), followed by the binary state transformer that implements the predicate functionality. If this state transformer is implemented naively (without updates), it would be blocking and unbounded. That is, each top-level element from the e_1 pipeline has to be cached until the predicate becomes true. This top-level element may span the whole stream and may become true at the end of the stream, thus making the predicate testing blocking and unbounded. More importantly, if we allow updates to the e_2 stream, then the predicate may become true at any point of time, which means that we would have to cache the entire e_1 stream. For example, in //book[author="Smith"], even if a book is not authored by Smith, it would still need to be cached, because later its author may be updated to Smith.

Our state transformer for $e_1[e_2]$ uses the show/hide events to show/hide e_1 elements based on the outcome of the predicate e_2. The state of the predicate state transformer includes a substream number, nid, which is different for each top-level element in e_1, the incoming element depth (as in the /tag step), the condition stream depth, cdepth, the outcome counter, which counts how many times the predicate has been set to true while reading the current top-level element (the predicate is true if outcome > 0), and the flags, fixed_true/fixed_false that indicate whether the predicate is true/false and we are absolutely sure that no future update can change it. When the fixed_true/fixed_false flag is set, then the current element is propagated/removed and is closed for further updates. This can only happen if the predicate outcome is fixed, that is, if there is a fixed top-level cData in the predicate stream that is not empty or all the top-level cData in the predicate stream are fixed and empty.

Since it works on two streams, the predicate state modifier is binary, that uses one transformer, F_1(e) to handle the data stream from e_1:

```
if e = sE {
  if depth = 1 {
    nid ← new_id(); outcome ← 0
    fixed_true ← false; fixed_false ← true
    return [ sM(e.id,nid), e ]
  }
  depth ← depth+1
} else if e = eE {
  depth ← depth-1
  if depth = 1                        // end of current element
    if fixed_true                     // certain to be true
      return [ e, eM(e.id,nid), freeze(nid) ]
    else if outcome > 0               // true, but may be revoked
      return [ e, eM(e.id,nid) ]
    else if fixed_false               // certain to be false
      return [ e, eM(e.id,nid), hide(nid), freeze(nid) ]
    else                              // false, but may be revoked
      return [ e, eM(e.id,nid), hide(nid) ]
  else return [ e ]
} else return [ e ]
```

and another, F_2(e), for the condition stream associated with the predicate e_2:

```
if e = sE
  cdepth ← cdepth+1
else if e = eE
  cdepth ← cdepth-1
else if cdepth = 0 and e = cD {
  fixed_false ← fixed_false ∧ e.text = "" ∧ fixed[e.id]
  if e.text ≠ ""
    if fixed[e.id]
      fixed_true ← true
    else outcome ← outcome+1
}
return [ ]
```

that is, if e_2 delivers a non-empty cData event, the predicate becomes true. We use a counter (outcome) instead of a boolean flag so that we can undo the predicate when the stream from e_2 is adjusted using Adjust(s1,s2):

```
n ← outcome
outcome ← n+(s2.outcome-s1.outcome)
if (n = 0) ≠ (outcome = 0)
  if outcome > 0
    return [ show(nid) ]
  else return [ hide(nid) ]
else return [ ]
```

Note that every top-level element from e_1 has its own substream id, and, thus, its own copy of the state. If there is an update to the stream e_2, all the subsequent states of the e_1 elements will be adjusted but only those affected by the update will have their outcome changed. A FLWOR where-clause is similar to an XPath predicate but its scope is an entire tuple, rather than a top-level XML element.

C. Descendant-Or-Self Steps

The XPath steps //* and //tag over recursive data (such as //part, where a part may contain other parts, etc, at any depth), are in general unbounded, but not blocking. For example, when applied over the XML element

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

the //* step returns the sequence:

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

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

(to simplify the coding, subelements are generated in postorder, instead of preorder). Generating this stream without updates would require to cache each element of depth 2 (each one of the two elements tagged b). If the element tagged b were the only element of depth 2 in the entire stream, it would have required to buffer the entire stream.

The reason that we needed a large state for //* was to insert the events of depth $d+1$ before the events of depth d, for every depth d. We can overcome this problem by generating all the nested elements at once, embedded inside some insert-before updates that move these elements to the correct place. More specifically, every event of nesting depth $d > 0$ is repeated $d-1$ times at the time it is received, without caching the event. The trick is that the events of depth $d+1$ are embedded inside an insert-before update that moves them before the latest element of depth d.

The state consists of the depth level and the mapping $m[\text{id}, d]$, which assigns a new unique id to the elements with a given id and depth d. The state modifier $F(e)$ is:

```
if e = sE {
  if depth > 0 {
    nid ← m[e.id,depth-1]
    if depth = 1
      return [ sM(e.id,nid), sE(nid,e.tag) ]
    else return { sE(m[e.id, i], e.tag) ‖ i ∈ [0 . . . depth-2] }
           ++[ sB(m[e.id,depth-2],nid), sE(nid,e.tag) ]
  }
  depth ← depth+1
} else if e = eE {
  depth ← depth-1
  if depth > 0 {
    nid ← m[e.id,depth-1]
    if depth = 1
      return [ eM(e.id,nid), eE(nid,e.tag) ]
    else return [ eE(nid,e.tag), eB(m[e.id,depth-2],nid) ]
           ++{ eE(m[e.id, i], e.tag) ‖ i ∈ [0 . . . depth-2] }
  }
} else if e = cD {
  if depth > 0
    return { cD(m[e.id, i], e.text) ‖ i ∈ [0 . . . depth-2] }
} else return [ e ]
```

eg, the XML element <a><c>x</c>:

```
[ sE(0,a),sE(0,b),sE(0,c),cD(0,x),eE(0,c),eE(0,b),eE(0,a) ]
```

becomes after //*:

```
[ sM(0,1), sE(0,b), sE(0,c), sB(0,1), sE(1,c), cD(0,x),
  cD(1,x), eE(1,c), eB(0,1), eE(0,c), eE(0,b), eM(0,1) ]
```

which is equivalent to <c>x</c><c>x</c>, after the updates are applied.

This transformer is inert because its state (m and depth) is invariant to updates. The descendant-of step, //tag, is similar to //* but requires that e.tag = tag and d be equal to the number of recursive elements with the same tag. That is, for non-recursive elements, no updates are generated, making //tag as efficient as /tag.

D. Sorting

Sorting, when implemented naively, is blocking and unbounded. We can unblock it by inserting each incoming element to the correct place by using an insert-after update. To achieve this, we maintain a mapping, keys, from ids to sorting keys. For each element with a sorting key, k, we insert it after the element that has the maximum key among all those with sorting key $< k$. (We assume that the sorting key is coming at a separate substream and is exactly one for each element; in general, if the sorting is on multiple keys, we would have multiple streams.) Unfortunately, the position to insert an element can only be determined after the key is extracted from the element, which can be at the end of the element in the worst case. Therefore, we use a queue to suspend the current element's events, which are released immediately after we get the key. The state modifier for the element sequence to be sorted is F_1(e):

```
if e = sS {
  nid ← new_id(); keys[nid] ← " "
  return [ e, sM(e.id,nid), eM(e.id,nid) ]
} else if e = sE {
  depth ← depth+1
  if depth = 1
    { clear queue; found_key ← false }
} else if e = eE {
  depth ← depth-1
  if depth = 0 and found_key
    return [ e, eA(mid,nid) ]
}
if not found_key
  queue.enqueue(e)
else return [ e ]
```

The stream with the sorting keys has the following state modifier F_2(e):

```
if e = cD {
  key ← e.text; nid ← new_id()
  keys[nid] ← key; found_key ← true
  let keys[mid] be the max key with keys[mid]<key
  return [ sA(mid,nid) ]++[ 'all events suspended in queue' ] }
```

Note that sorting is now non-blocking but it still requires unbounded state.

E. Backward Steps

XPath backward axis steps are very difficult to implement efficiently in stream processing. They potentially require an unbounded state. An XQuery 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 immediately after it is generated (that is, before it is passed through the pipeline). The reason for doing so is that one can potentially reach the root through a number of backward steps, and from the root, one can reach any element in the entire stream, including those that have already passed through. Cloning is easy and does not require any caching: each event is repeated twice under different substream numbers. Then, the state transformer that implements a backward axis is a special join between the incoming stream and the

cloned stream source. Here we show how the XPath steps ancestor::* and parent are implemented in our framework.

Immediately before the state transformer that implements ancestor::*, the cloned source is passed through the state transformer for the //* step (Section VI-C), that is, each element of the cloned source at depth d is repeated $d-1$ times. The state modifier that implements ancestor::* is very similar to that for a predicate (Section VI-B). The only difference is that we now have two variables, left_end and right_end, that hold the latest eE event (at any depth) of the cloned source and the top-level eE (of zero depth) of the incoming stream, respectively. When these two events are identical, then the cloned source element is an ancestor of the incoming element. To simplify the code, in the following state modifiers, we have removed the code for fixed_true/fixed_false that removes the state when the outcome is fixed. The state modifiers, $F_1(\mathsf{e})$, for the cloned stream source that has already passed through the //* step is:

```
if e = sE {
  depth ← depth+1
  if depth = 1 {
    nid ← new_id()
    left_end ← ∅; right_end ← ∅; outcome ← 0
    return [ sM(e.id,nid), e ]
  }
} else if e = eE {
  depth ← depth-1
  left_end ← e                                   (1)
  if e = right_end                               (2)
    outcome ← outcome+1                          (3)
  if depth = 0
    if outcome > 0
      return [ e, eM(e.id,nid) ]
    else return [ e, eM(e.id,nid), hide(nid) ]
}
return [ e ]
```

while the other, $F_2(\mathsf{e})$, for the the ancestor::* input increments outcome when e=left_end (this is an OID equality, which is accomplished for SAX events with an extra event parameter OID, which is set at the source). The Adjust(s1,s2) method is exactly the same as that of a predicate (Section VI-B). The parent axis step (/..) works like the ancestor::* step, but the statements (1)-(3) are done at depth=1 only.

VII. Performance Evaluation

XFlux is in its early stage of implementation, although many essential XQuery features have already been implemented, including XQueries over single input streams, all forward and some backward XPath steps, concatenation, construction, predicates, sorting, and some aggregations. The translation of XQuery to state transformer pipelines is straightforward and described in our earlier work [4]. XFlux is available at http://lambda.uta.edu/xflux/. The platform used for our experiments was a 3GHz Pentium 4 processor with 1GB memory on a Linux PC. The code was written in Java (J2RE 1.5.0) and used a SAX parser (Piccolo). We used one artificially generated (XMark) and one real (DBLP) dataset:

Benchmark	document	size	events	time
XMark	X	224 MB	12.7 M	9.6 secs
DBLP	D	318 MB	31.3 M	18.6 secs

where events is the number of SAX events in millions and time is the time in seconds used to tokenize the document using SAX. The goal of this evaluation was to assess the impact of our method of generating updates during query execution to unblock operations and to reduce buffering. Therefore, in our performance evaluation, the input stream was simply the tokenized XML document (X or D) without any incoming updates. We also compared our system performance with that of SPEX [16], which is freely available and is a good representative of the automata-based systems. Since automata-based systems have been shown to be optimal for a restricted subset of XPath (with simple predicates and without backward steps), it would be important to see how our system compares to SPEX for these restricted queries. We have not compared our system with non-streaming XQuery processors, such as Galax and Saxon, because these systems generally show a slow first response time (since they have to parse the entire XML document before they start processing) and cannot handle XML documents larger than few hundred MBs without indexing the documents first. Thus, although these systems do an excellent job on indexing and processing XQueries on stored data, they are not suitable for continuous queries over stream data arriving at high rates.

We used nine benchmark queries that cover most of the techniques described in our paper:

```
1) X//europe//item[location="Albania"]/quantity
2) X//item[location="Albania"][payment="Cash"]/location
3) X//*[location="Albania"]/quantity
4) count(X//item[location="Albania"]/..)
5) count(X//item[location="Albania"]/ancestor::europe)
6) count(X//item[location="Albania"]/ancestor::*//location)
7) <result>{
     for $c in X//item
     where $c/location = "Albania"
     return <item>{ $c/quantity, $c/payment }</item>
   }</result>
8) D//inproceedings[author="John Smith"]/title
9) for $d in D//inproceedings
   where contains($d/author,"Smith")
   order by $d/year
   return ($d/year/text(),": ",$d/title/text(),"\n")
```

We got the following measurements from these queries:

Q	XFlux time	MB/s	SPEX time	events	mem
1	16 secs	14.0	52 secs	17 M	452 KB
2	35 secs	6.4	42 secs	89 M	683 KB
3	197 secs	1.1	70 secs	683 M	412 KB
4	116 secs	1.9	-	326 M	854 KB
5	33 secs	6.8	-	95 M	487 KB
6	124 secs	1.8	-	329 M	466 KB
7	29 secs	7.7	-	71 M	779 KB
8	84 secs	3.8	113 secs	231 M	561 KB
9	92 secs	3.5	-	194 M	790 KB

where the XFlux/SPEX time is the execution time in seconds for XFlux/SPEX, MB/s is the throughput in MB/secs, events is the number of state transformer method calls in millions, and mem is the memory used in Java. A dash means that SPEX does not directly support the evaluation of the query. We can see that for query 3, SPEX is far faster than ours, because, in

contrast to automata-based systems, we translate XQuery one-step-at-a-time, so that our XQuery translation is compositional and general. That is, in our framework, the //* step is translated without any knowledge about the XPath steps that follow. The backward steps in queries 4, 5, and 6 resulted in an acceptable overhead, comparable to other queries.

VIII. Related Work

Most current methods for processing XPath queries as well as general XQueries over XML streams are based on finite state machines and transducers augmented with buffers [11], [9], [8], [12]. These approaches do an excellent job on the stream processing of simple XPath queries, but their extensions to handle general predicates [9], [12] and complex XQueries [11] turned out to be hard. One reason for this is that XQuery was designed as a functional language, where XQuery expressions may appear at any place in a query, thus suggesting a compositional translation of queries. On the other hand, the approaches based on finite state machines require a holistic view of an XPath expression to construct and optimize the automaton. In addition, after the 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 [11]. This is very hard even for the simplest XQuery constructs, such as sequence concatenation and element construction. A notable exception that avoids the pitfalls of integrating these two incompatible models (automata and the functional paradigm) is the Raindrop project [17], which decomposes a query into two parts: one made out of automata that processes all the paths in a query and another based on algebraic techniques that processes the rest of the query. Another hybrid approach that incorporates streaming techniques into an existing stored XML database system (Galax) is by Fernandez, et al [5]. It uses operators to materialize XML trees from stream sections and to streamline XML trees. Unlike our work, it processes recursive queries by copying stream sections attached to multiple nested cursors, which requires unbounded space in the worst case.

In an earlier work, we have developed a pull-based streaming processor for XQuery, called XQPull [4], that provides methods for reducing buffering for general predicates, forward and backward axes. It is based on retarded streams, which allow multiple and nested streams to be interleaved in the same physical stream. An element of an inner stream must be placed after the enclosing element from the outer stream but this placement is delayed to the end of the query evaluation, where both elements are expected to be smaller. In addition, each XQPull iterator that needs to postpone computations to avoid buffering must introduce events tailored to this particular operator exclusively, and all the other operators must handle these new special events gracefully. This makes XQPull hard to extend with new lazy iterators. In our framework, on the other hand, the special events are fixed and, more importantly, they can be handled by a fixed operator wrapper without burdening the operators by requiring to include extra code to handle these events. This effectively simplifies coding and facilitates extensions to include other lazy operators and techniques. Finally, unlike XQPull, our framework can handle incoming updates, thus facilitating processing of update streams.

Our work is also related to incremental view maintenance where updates arrive at very high rates [14]. The traditional view maintenance research is focused on generating incremental updates on stored data while ours is focused on generating updates on the actual query output (since we do not store data on secondary storage). Furthermore, in addition to handling incoming updates, which is the goal of incremental view maintenance, we let stream operators generate updates in order to reduce buffering and improve performance.

IX. Conclusion and Future Work

The main theme of this paper was to introduce updates during query processing that encapsulate the blocking or unbounded parts of the operations, so that, when performed at the end, they will complete the functionality of the operation. The reason for this lazy evaluation is that later is often better than now, since casual queries are expected to produce more disperse result streams than the input streams, thus reducing the caching requirements. In the future, we will make our system be able to perform some of these updates earlier, if state maintenance and adjustment become more expensive than evaluating the actual updates (based on some QoS metrics).

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

References

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *PODS'02*.
[2] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive Caching for Continuous Queries. In *ICDE 2005*.
[3] S. Bose and L. Fegaras. Data Stream Management for Historical XML Data. In *SIGMOD'04*.
[4] L. Fegaras, R. Dash, and Y. Wang. A Fully Pipelined XQuery Processor. In *XIME-P'06*.
[5] M. Fernandez, et al. XQuery Streaming *à la Carte*. In *ICDE'07*.
[6] D. Florescu, et al. The BEA Streaming XQuery Processor. VLDB Journal 13(3): 294-315 (2004).
[7] L. Golab and M. T. Ozsu. Issues in Data Stream Management. In *SIGMOD Rec.*, 32(2):5–14, 2003.
[8] T. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. In *ICDE'03*.
[9] A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *SIGMOD'03*.
[10] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *SIGMOD'97*.
[11] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *VLDB'02*.
[12] F. Peng and S. Chawathe. XPath Queries on Streaming Data. In *SIGMOD'03*.
[13] V. Raman and J. M. Hellerstein. Partial Results for Online Query Processing. In *SIGMOD'02*.
[14] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay. How To Roll a Join: Asynchronous Incremental View Maintenance. In *SIGMOD'00*.
[15] SAX. At http://www.saxproject.org/.
[16] SPEX: XPath Evaluation Against XML Streams. At http://spex.sourceforge.net/.
[17] H. Su, E.A. Rundensteiner, and M. Mani. Semantic Query Optimization in an Automata-Algebra Combined XQuery Engine over XML Streams. In *VLDB'04*.