Processing Continuous Historical Queries over XML Update Streams

Leonidas Fegaras

University of Texas at Arlington, CSE
416 Yates Street, P.O. Box 19015
Arlington, TX 76019

fegaras@cse.uta.edu

ABSTRACT

We address the problem of processing continuous historical 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 to trees. Our query language for expressing historical queries is XQuery extended with temporal annotations that specify sliding windows that focus on the stream results of query subexpressions, rather than on the query input streams. We have developed a novel architecture that can evaluate historical queries over large XML data streams using short history lists. Instead of buffering a fixed-size window, sliding through the input stream events, we focus on the temporal requirements of individual operations derived from the query and use history lists to cache the state history of the operation as long as necessary to answer the query.

1. INTRODUCTION

Although most current data stream management systems (DSMSs [2]) have focused on numerical and relational data streams, there is a recent interest in processing stream data in XML form. 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 data streaming. Furthermore, there is a large number of emerging streaming applications that use XML as a data stream format, since it is now the language of choice for communication between co-operating systems. Examples of such applications include the on-the-fly query processing of XML documents using push-based (SAX [19]) or pull-based (StAX [20]) stream processing, retrieving and integrating results from web services (based on the SOAP and WSDL standards), content-based XML routing for selective dissemination of information in publisher-subscriber systems, processing stock market updates and real-time news feeds (RSS), and extracting XML descriptors and description schemes (DDL) embedded in MPEG-7 streams.

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 [2]. When data are transmitted in streams of tuples, the unit of a relational stream is unquestionably a tuple. For relational DSMSs, the stream is typically an infinite sequence of tuples, 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. But when applied to XML data, 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. Currently, the most common method for XML fragmentation is XML tokenization, popularized by the SAX API for XML [19], which breaks the structure of an XML tree down into a series of linear events or tokens that can be transmitted in a stream. Since most common sources of XML data, such as tokenized XML documents, result to finite streams, there is no general agreement on what an infinite, continuous XML stream should look like. One way to generate an infinite stream is to let one top-level XML element to be open-ended, thus allowing an infinite number of children to be appended to this element. This solution is limited, since it does not allow continuous additions to multiple elements (such as, continuous updates to both stocks and mutual funds). More importantly, it does not distinguish between new additions and updates, a piece of information that can be very useful when expressing continuous historical queries. For example, one may want to find which stock increased its value by at least 10% since the last update. This is a continuous query that produces output notification every time a new stock update comes in the input stream whose value is at least 10% higher than its old value. To answer this query, for each incoming update in the stream, one needs to find the stock price it replaces. If this information is not explicitly given in the stream (as a direct link between the new and the updated data), it must be specified as part of the query as a correlation based on con-

tent, which makes the query unnecessarily complicated. We propose a better and more precise solution: an XML stream in our framework is a finite stream made out of XML tokens followed by (or interleaved with) an infinite stream of well-formed updates. We have defined new types of tokens, not present in regular tokenized XML documents, that allow one to express well-formed updates to XML data. A well-formed update is a finite sequence of tokens that corresponds to a complete XML node, when this sequence is materialized into a tree. It specifies the insertion, deletion, or replacement of an XML node that has already passed through the stream as another sequence of tokens. That is, both the source and the destination of an update correspond to complete XML nodes. Furthermore, an update specifies exactly which node to replace, thus explicitly correlating the past with the current values in the data stream itself, rather than requiring this information to be expressed in the query. That way, updates to streamed XML nodes can form history lists that are easy to query using a temporal query language. Each history list associated with an update can be seen as a focused sliding window specific to this update only.

Our framework for stream processing is based on the idea of a *state transformer*, which generalizes earlier techniques based on finite state machines or transducers augmented with buffers [16, 13, 12, 17]. A state transformer operates on a stream one-token-at-a-time, using a state to keep information between calls. It is simply a function from a single XML token to a sequence of tokens, which is called repeatedly on input tokens. In an earlier work [7], we have developed a framework that maps a subset of XQuery [21] to query evaluation pipelines that consist of state transformers exclusively. Based on compositional translation techniques, each XQuery syntactic structure is mapped to a simple state transformer, while a given XQuery is mapped to a pipeline by composing these state transformers in the same way the corresponding XQuery syntactic structures are composed to form the query. Our query language for expressing historical queries is XQuery extended with temporal annotations that specify sliding windows that focus on the stream results of query subexpressions, rather than on the query input streams, as is done by earlier work [22]. These annotations allow us to treat historical data differently, based on the temporal requirements of the query, which offers many opportunities for memory reduction.

The main goal of our framework is to handle the incoming stream with the embedded updates without any blocking, so that an update in the input stream would cause an immediate update to the output display. Therefore, it is crucial to design the output medium in such way that would reflect the updates as if these updates were applied to the input stream directly and the query were re-evaluated continuously. But it would be highly inefficient to re-evaluate the entire query every time a new update is received in the stream and it would definitely be undesirable to update the actual input stream because then we would have to cache an arbitrarily long sliding window. 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. Therefore, what we want is, instead of eagerly performing the updates on cached portions of the stream, to propagate the updates through the query evaluation pipeline without any delay, all the way to the result display, which prints the query answers. That way, the result display will print the query results continuously, replacing old results with new. For example, the query that prints the stock quotes whose value increased by at least 10% will show the results continuously on the answer display as if the query were evaluated continuously over the entire stream, starting from the beginning of the stream on each update and re-evaluating the query from scratch each time. But what really happens in our framework is that the evaluation is done incrementally, one-event-at-a-time (in a way reminiscent to incremental view maintenance [18]). The updates are propagated through the query pipeline all the way to the answer display, which is an editable text window, and cause updates to the actual display text. That is, quotes may be erased or inserted in the display continuously, based on the incoming quotation updates.

But the propagated updates may cause state changes to the state transformers in the pipeline. For example, the state transformer that counts XML elements and sends updates to the output when the count changes, must include a counter in its state. When elements that have already passed through the stream are removed or new elements are inserted, due to some embedded update, the counter must decremented/incremented to reflect this update. We address this problem by adjusting the state of a state transformer based on the incoming updates. Given a state transformer that accepts regular data streams without updates, our framework automatically generates 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. When a new update is received by a state transformer, it causes a state transition and, normally, the new state replaces the old (if there is no temporal dimension to the query). For historical queries, old states are kept in history lists attached to state transformers. We have developed a novel architecture that can evaluate historical queries over large XML data streams using the shortest history lists possible. Instead of buffering a fixed-size window, sliding through the input stream events, we focus on the temporal requirements of individual operations derived from the query and use history lists to cache the state history of the operation as long as necessary to answer the query. That is, instead of caching input events and use them to go back and forth in history, we cache states only, which often have a smaller total memory footprint than the minimal sliding window required to answer the same query.

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

- The units of our XML streams are SAX-like tokens that allow one to express both regular XML data and well-formed, continuous, cascaded updates to these data that facilitate the processing of long-running, continuous queries.

- Based on this update stream data model, we propose some very simple syntactic extensions to XQuery that allow one to express very sophisticated historical continuous queries, by specifying time- or version-based

sliding windows over the results of query subexpressions, rather than over the input streams.

- 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. That way, the result display prints the query results 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. By analyzing the temporal characteristics of the running query, our framework can precisely predict the amount of state history needed by each pipeline operator to answer the query. This analysis results to a finer control of buffering than by methods based on a single global sliding window.

The paper is organized as follows. Section 2 presents our formalism for XML update streams. Section 3 defines our temporal extensions to the XQuery syntax. Section 4 lists five examples of temporal queries and discusses some of the challenges to be addressed when processing these queries. Section 5 gives the semantics of our update streams and query extensions. The semantics analysis is not used by the subsequent sections so the reader may safely skip this section. Section 6 describes the state transformers, which are the unit components of our query evaluation pipelines. Section 7 describes our framework for wrapping each state transformer to make it capable of storing and using history lists, if it is required by the query subexpression associated with this state transformer. Section 8 describes the implementation of the state transformer wrapper. Section 9 describes the most important state transformer, common to all query evaluation pipelines, namely the result display, which displays the results continuously, replacing old results with new. Finally, Section 10 discusses related work.

2. XML UPDATE STREAMS

In our framework, a regular 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)[1]:

startStream:	sS(stream,id)
endStream:	eS(stream,id)
startElement:	sE(id,tag)
endElement:	eE(id,tag)
characterData:	cD(id,text)

Our query processing framework requires multiple virtual streams embedded into the same physical stream. Each virtual stream has a unique stream number, as indicated in the sS/eS, which mark the beginning/ending of each virtual stream. Each event e has a unique id, e.id, associated with a single stream. The necessity for having multiple ids for a stream will be apparent next, when we describe updates that may contain interleaving regions of events. The mapping stream[id] gives the stream number associated with a given id. The events sE, eE, and cD correspond to the well-known SAX events [19] for the beginning and the end of an XML element, and for a text node (for simplicity, element attributes are ignored). For example, the stream that consists of the XML element <name>Smith</name> may be tokenized into the event sequence:

[sS(10,3), sE(3,"name"), cD(3,"Smith"),
eE(3,"name"), eS(10,3)]

that is, the stream number is 10 and all events have id=3, with stream[3]=10.

The well-formedness, $v \in \mathcal{WF}_i$, of a sequence $v \in \mathcal{E}^*$ (ie, a list of $\mathcal{E}$ elements) for an id 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 $v \in \mathcal{WF}_i$ associated with the id i must be properly nested while the events with a different id are irrelevant (first rule) and should be allowed in $\mathcal{WF}_i$.

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

startMutable:	sM(i,j)	endMutable:	eM(i,j)
startReplace:	sR(i,j)	endReplace:	eR(i,j)
startInsertBefore:	sB(i,j)	endInsertBefore:	eB(i,j)
startInsertAfter:	sA(i,j)	endInsertAfter:	eA(i,j)

Let sU be the starting event of an update (that is, it can be an sM, sR, sB, or sA event) and eU be the matching end of sU (that is, eM, eR, eB, or eA, respectively). A sequence $[\text{sU}(i,j)] +\!\!+ v +\!\!+ [\text{eU}(i,j)]$ defines a substream sequence with id j embedded in the substream sequence with id i, so that stream[j] = stream[i] and $v \in \mathcal{WF}_j$, that is, the events in v associated with the id j are well-formed. In particular, the mutable sequence $[\text{sM}(i,j)] +\!\!+ v +\!\!+ [\text{eM}(i,j)]$ defines a subsequence of the sequence with id i that consists of events with id 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 with id=i or to an earlier update with id=i. That is, updates too are open for updates. An update id may be used multiple times for performing cascaded updates. Since we are interested in temporal queries, a replace update does not necessarily remove the previous value; instead it forms a history list that consists of as many replaced versions as necessary to answer the current temporal query (described in the next section). 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.

For example, the sequence

[sS(10,0), 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), eS(10,0)]

[1]Although this section summarizes our earlier work on streams [7], no familiarity with this work is assumed.

defines a mutable region with id=1, embedded in the subsequence with id=0 of the stream 10 and contains the text "x", followed by an update to this region (a replacement with id=2), that 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:

[sS(10,0), cD(0,"w"), cD(0,"y"), cD(0,"z"), eS(10,0)]

3. TEMPORAL EXTENSIONS TO XQUERY

The language of choice for querying XML data is now XQuery [21], which has replaced XPath as the standard query language for XML. In contrast to XPath, which supports data navigation and filtering only, XQuery allows arbitrary nesting of query expressions, user-defined functions, concatenation, element construction, sorting, and joins, which are very difficult to streamline efficiently. We extend the XQuery syntax with temporal language constructs for effective querying of historical data. Our extensions are based on our earlier work on XCQL [4], which has been influenced by relational temporal languages and by continuous query languages based on sliding windows, such as CQL [22, 15]. Instead of restricting the lifespan of input stream data using a single sliding window, as is done in CQL, we let our temporal annotations apply to any XQuery expression, thus providing a finer control, which, as we will see, may result to more opportunities for memory reduction. The syntactic extensions to XQuery are:

$e\#v$	version projection	(the past vth version)
$e?t$	time projection	(the version before t secs)
$e\#[v]$	version sliding window	(all versions from 0 to v)
$e?[t]$	time sliding window	(all versions in the last t secs)

where e is an arbitrary XQuery expression, the integer $v \geq 0$ is a version number, and the real number $t \geq 0$ is elapsed time in seconds (the current time has always 0 elapsed time). The version projection $e\#v$ returns the vth version, for $v \geq 0$, or () if this version does not exist, where the current version is always 0, the previous version is 1, etc. Without a projection, the value of e is equal to the current version, $e\#0$. The time projection $e?t$ returns the value of e at t seconds before the current time, or () if this version does not exist. That is, the time projection $e?t$ is the version $e\#v$ that has the minimum timestamp greater than or equal to the current time minus t. The time t can be any time duration, even years, but, of course, for a t longer than the elapsed running time of the continuous query, the $e?t$ value will be (). As we will see, it does not matter how many events have passed through the query engine; only the states relevant to the running query are cached.

For example, given the following stream subsequence:

```
   [ sM(1,2),
         sE(2,a),
t1            sM(2,3), sE(3,b), cD(3,X), eE(3,b), eM(2,3),
t2            sR(3,4), sE(4,b), cD(4,Y), eE(4,b), eR(3,4),
t3            sR(3,4), sE(4,b), cD(4,Z), eE(4,b), eR(3,4),
         eE(2,a),
     eM(1,2),
     sR(2,5),
t4       sE(5,a), sE(5,b), cD(5,W), eE(5,b), eE(5,a),
     eR(2,5) ]
```

when we evaluate the snapshot query ./a/b against this stream, we will first see X on the result display at time t_1, which will be replaced by Y at t_2, then by Z at t_3, and finally by W at t_4. If we evaluate the temporal query ./a#1/b, we will only see Z at t_4. Similarly, if we evaluate the temporal query ./a#1/b#2, we will only see Z at t_4. On the other hand, for the query ./a/b#1, we will see X at t_2, replaced by Y at t_3, and finally cleared at t_4.

The version sliding window $e\#[v]$ returns the sequence of the latest v versions. It is equivalent to $(e\#0, e\#1, \ldots, e\#v)$, that is, it uses sequence concatenation to return the first v versions. For example, if we evaluate the temporal query ./a/b#[1] over the previous stream subsequence, we will see X at t_1, replaced by YX at t_2, by ZY at t_3, and by W at t_4. The time sliding window $e?[t]$ returns the sequence of all the values of e during the last t seconds (including the current value). It is equivalent to $(e\#0, e\#1, \ldots, e\#v)$ such that the timestamp of $e\#v$ is greater than or equal to the current time minus t.

4. EXAMPLES OF TEMPORAL QUERIES

The following are examples of continuous XQueries over update streams:

1. Query 1 is a snapshot query that displays the titles and prices of all books published by Wiley and authored by Smith, sorted by their prices:

   Query 1:

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

   The qualified books (ie, those books that satisfy the query conditions) are displayed in the query result display continuously: when the first qualified book is found, 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 move 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.

Note that we have presented all the details of unblocking the sorting and other blocking XQuery operations in an earlier work [7]. This is accomplished by letting the pipeline stages associated with these operations generate updates too. These updates are handled in the same way as the updates embedded in the incoming stream. For example, counting elements is unblocked by sending repeated updates on the counter values, which continuously replace the answer in the result display.

2. Query 2 is a temporal query that displays all stocks whose quotation increased at least 10% since last time, sorted by their rate of change:

Query 2:

```
for $q in stream("tickers")//ticker
where $q/quote ≥ $q/quote#1 * 1.1
order by ($q/quote - $q/quote#1) div $q/quote
return <quote>{ $q/name, $q/quote }</quote>
```

In this query, $q/quote is the current quotation while $q/quote#1 is the previous one. As in Query 1, quotes may be erased, inserted, or move up or down in the display continuously, based on the incoming quotation updates. It does not matter if the updates come infrequently, say once a day; if the query is run continuously, only the latest pairs of quotes would be cached.

3. Suppose that a network management system receives two streams from a backbone router for TCP connections: one for SYN packets and another for ACK packets that acknowledge the receipt. Query 3 identifies the misbehaving packets that, although they are not lost, their acknowledgment comes more than a minute late:

Query 3:

```
for $a in stream("ack")//packet
where not (some $s in stream("syn")//packet?[60]
           satisfies $s/id = $a/id
                 and $s/srcIP = $a/destIP
                 and $s/srcPort = $a/destPort)
return <warning>{
          $a/id, $a/destIP, $a/destPort
       }</warning>
```

That is, for each packet from the ack stream, the query looks 60 seconds back in the packet history at the syn stream. It is interesting to note why we took the arrival of ACK as the reference point (current time) rather than that of SYN. Basically, the most convenient reference point is the latest time among all events, since we cannot look at the future.

4. In a radar detection system, a sweeping antenna monitors communications between vehicles by detecting the time of the communication, the angle of the antenna when it captures the signal, the frequency, and the intensity of the signal. This information is streamed to a vehicle monitoring system. Query 4 locates the position of a vehicle by joining the streams of two radars over both frequency and time and by using triangulation:

Query 4:

```
for $r in stream("radar1")//event?[1],
    $s in stream("radar2")//event?[1]
where $r/frequency = $s/frequency
return <position>{
          triangulate($r/angle,$s/angle)
       }</position>
```

where the function triangulate uses triangulation to calculate the x-y coordinates of the vehicle from the x-y coordinates of the two radars and the two sweeping angles. Here, we assume that each antenna rotates at a rate of one round per second, thus, when a vehicle is detected by both radars, the two events must take place within one second.

5. Some vehicles, such as buses and ambulances, have vehicle-based sensors to report their vehicle IDs and locations periodically. In addition, road-based sensors report their sensor IDs and the speed of the passing vehicles. Some traffic lights, on the other hand, not only report their status each time it changes, but they also accept instructions to change their status (e.g., from red to green). When an ambulance is close enough to a traffic light, Query 5 returns events that switch the light to green at a time that depends on the speed and the position of the ambulance:

Query 5:

```
for $v in stream("vehicle")//event[type="ambulance"]?[1],
    $r in stream("road_sensor")//event?[1],
    $t in stream("traffic_light")//event?[1]
where distance($v/location,$r/location) < 0.1
   and distance($v/location,$t/location) < 10
return <set id="{$t/id}">
          <status>green</status>,
          <time>{ (distance($v/location,$t/location)
                   div $r/speed) }</time>
       </set>
```

5. SEMANTIC ANALYSIS

In this section, we provide a semantic analysis of our update streams and query extensions. This analysis is given for completeness only. It is not used by the subsequent sections so the reader may safely skip it.

Each event $e \in \mathcal{E}$ in the stream $S \in \mathcal{E}^*$ is assigned a timestamp $\mathcal{T}[e] \in \Re$, which is the time in seconds when this event was received. Let $\mathcal{CT}$ be the timestamp of the current time. The valid lifespan of an event e, $\mathcal{V}[e] \in \Re^2$, is the validity interval of the event relative to the current time. It is initially equal to $[0, \infty)$, which indicates that it is valid since the beginning of time (∞) up to and including the current time (0).

The meaning of a stream $S \in \mathcal{E}^*$ is another stream $S' \in \mathcal{E}^*$ without update events (i.e., S' has only sS, eS, sE, eE, and cD events) that is equivalent to S at the current time. The valid lifespan mapping of S' is $\mathcal{V}'$. The following rules extract S' from S and calculate V' incrementally. The first two rules convert the insertBefore/insertAfter updates into mutable regions by moving the updating events before/after

the updated region:

$$[\mathsf{sM}(i,j)]{+\!\!+}v_1{+\!\!+}[\mathsf{eM}(i,j)]{+\!\!+}v_2{+\!\!+}[\mathsf{sB}(j,k)]{+\!\!+}v_3{+\!\!+}[\mathsf{eB}(j,k)]$$
$$\rightarrow\ [\mathsf{sM}(i,k)]{+\!\!+}v_3{+\!\!+}[\mathsf{eM}(i,k),\mathsf{sM}(i,j)]{+\!\!+}v_1{+\!\!+}[\mathsf{eM}(i,j)]{+\!\!+}v_2$$

$$[\mathsf{sM}(i,j)]{+\!\!+}v_1{+\!\!+}[\mathsf{eM}(i,j)]{+\!\!+}v_2{+\!\!+}[\mathsf{sA}(j,k)]{+\!\!+}v_3{+\!\!+}[\mathsf{eA}(j,k)]$$
$$\rightarrow\ [\mathsf{sM}(i,j)]{+\!\!+}v_1{+\!\!+}[\mathsf{eM}(i,j),\mathsf{sM}(i,k)]{+\!\!+}v_3{+\!\!+}[\mathsf{eM}(i,k)]{+\!\!+}v_2$$

for all non-negative integers i, j, k, and for every $v_1 \in \mathcal{WF}_j$, $v_2 \in \mathcal{WF}_i$, and $v_3 \in \mathcal{WF}_k$. These rules do not change the valid lifespans $\mathcal{V}$ of the events. A replacement update is converted in the same way as an insertBefore update:

$$[\mathsf{sM}(i,j)]{+\!\!+}v_1{+\!\!+}[\mathsf{eM}(i,j)]{+\!\!+}v_2{+\!\!+}[\mathsf{sR}(j,k)]{+\!\!+}v_3{+\!\!+}[\mathsf{eR}(j,k)]$$
$$\rightarrow\ [\mathsf{sM}(i,k)]{+\!\!+}v_3{+\!\!+}[\mathsf{eM}(i,k),\mathsf{sM}(i,j)]{+\!\!+}v_1{+\!\!+}[\mathsf{eM}(i,j)]{+\!\!+}v_2$$

but with new lifespans: Let $[t_1,t_2) = \mathcal{V}[\mathsf{sM}(i,j)]$ and $t_r = \mathcal{CT}{-}\mathcal{T}[\mathsf{sR}(j,k)]$. Then each event e in $[\mathsf{sM}(i,k)]{+\!\!+}v_3{+\!\!+}[\mathsf{eM}(i,k)]$ has a new lifespan $\mathcal{V}'[e] = [t_1,t_r)$, while each event e in $[\mathsf{sM}(i,j)]{+\!\!+}v_1{+\!\!+}[\mathsf{eM}(i,j)]$ has a new lifespan $\mathcal{V}'[e] = [t_r,t_2)$. That is, this replacement update introduces a new version of data that is valid at an interval immediately before the interval of the updated value.

The next step is to remove all sM/eM from the stream, resulting to a stream S' without update events. All the information required to process any temporal XQuery is now incorporated into the new lifespan mappings $\mathcal{V}'$. For example, the snapshot view of S' is the substream consisting of all events e with $0 \in \mathcal{V}'[e]$, that is, all the events that are valid at current time. The new event sequence is translated into XML code, where the lifespan information is incorporated into every element as attribute values ST/ET (for starting/ending times):

$$XML([\mathsf{cD}(i,\text{text})]{+\!\!+}v) \ =\ (\text{text},\ XML(v))$$
$$XML([\mathsf{sE}(i,\text{tag})]{+\!\!+}v_1{+\!\!+}[\mathsf{eE}(i,\text{tag})]{+\!\!+}v_2)$$
$$=\ (\text{<tag ST} = t_1\ \text{ET} = t_2\text{>}\ XML(v_1)\ \text{</tag>},\ XML(v_2))$$

where $(t_1,t_2) = \mathcal{V}'[\mathsf{sE}(i,\text{tag})]$ and comma is XML sequence concatenation.

We now give the semantics of our temporal extensions to XQuery. To facilitate the semantic analysis, we modify every XQuery so that if there is an XQuery subexpression e in the query that does not have a temporal extension (i.e., it is not one of $e\#v$, $e?t$, $e\#[v]$, or $e?[t]$) and is used directly in a comparison, a concatenation, an element construction, or a function argument, then it is replaced with $e\#0$ (i.e., the latest value of e). Then, assuming that the query operates over $XML(S')$, the temporal extensions are simply compiled away into pure XQuery code as follows:

$$\begin{array}{rcl} e\#v & = & e[v+1] \\ e?t & = & e[@ST \le t \text{ and } t<@ET] \\ e\#[v] & = & e[\text{position} \le v+1] \\ e?[t] & = & e[@ST \le t] \end{array}$$

where position is the context position in XPath. That is, a version projection/window is translated into an XPath positional restriction and a temporal projection/window is translated into an XPath predicate.

This process of materializing the updates in the stream, then transforming the stream into annotated XML data, and finally evaluating the continuous query over the resulting XML, must be done every time a new update is received, and, thus, it is terribly inefficient. But the purpose of this process is to give well-defined semantics to assert the correctness of our real evaluation framework. The actual evaluation is done incrementally, one-event-at-a-time, without materializing any XML data (only states are cached in memory). The correctness of any streamline implementation that implements a query Q using a stream engine R is established by proving $Q(XML(\mathit{Transform}(S))) = XML(\mathit{Transform}(R(S)))$, for any well-formed stream S, where *Transform* is the above algorithm that transforms S into S' without update events. We leave this correctness proof to a future work.

6. STATE TRANSFORMERS

In our framework, an XQuery evaluation engine is implemented as a pipeline of stages, where each stage implements a simple XQuery operation, such as an XPath step. Each pipeline stage is associated with a tuple $(\mathcal{S}, s, z, k, f)$ that contains a state type $\mathcal{S}$, a state s of type $\mathcal{S}$, an initial state z of type $\mathcal{S}$, a stream number k, and a *state transformer* f of type $\mathcal{E}\times\mathcal{S} \to \mathcal{E}^*\times\mathcal{S}$. The *effective state transformer* f_k applies f to any event e with stream[e.id]=k, while returning the event for all others:

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

The *sequence transformer* f_k^* of type $\mathcal{E}^*\times\mathcal{S} \to \mathcal{E}^*\times\mathcal{S}$ is defined recursively:

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

A sequence transformer f_k^* is *inert* iff $\forall s \in \mathcal{S},\ \forall i$:

$$\mathsf{stream}[i] = k,\ \forall v \in \mathcal{WF}_i,\ \exists v' \in \mathcal{E}^* :\ f_k^*(v,s) = (v',s)$$

that is, if f_k^* does not change the state for any well-formed sequence of the input stream k. Most XQuery operations, including most XPath steps, correspond to inert state transformers. Counting XML elements is an example of a non-inert transformer since its state must include a counter that increments at each XML element.

For example, the XPath step /tag (the child-of axis) has state (depth,pass) of type int×boolean, an initial state value (0,false), and is associated with the following state transformer $f(e,s)$:

```
if e = sE                          // if e is a startElement
   then if s.depth = 1
      then if e.tag = "tag"    // switch to passing mode
         then return ( [ e ], (s.depth+1,true) )
      else return ( [ ], (s.depth+1,false) )
   else return ( if s.pass then [ e ] else [ ],
                 (s.depth+1,s.pass) )
else if e = eE
   then if s.depth = 2      // switch to discard mode
      then return ( if s.pass then [ e ] else [ ],
                    (s.depth-1,false) )
      else return ( if s.pass then [ e ] else [ ],
                    (s.depth-1,s.pass) )
else return ( if s.pass then [ e ] else [ ], s )
```

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

One way of 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_k(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_k 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 (also known as the iterator model in database query processing), where an iterator reads (pulls) as many tokens from the input as necessary to produce a single tokens for the next iterator in the pipeline [20]:

```
class F extends Iterator {
    input: Iterator;
    state: S ← z;
    queue: E* ← [ ];
    function next () : E {
        while queue = [ ] do
            (queue,state) ← f_k(input.next(),state);
        e ← queue.head();
        queue ← queue.tail();
        return e } }
```

Although the pull-based model is very effective in traditional database query processing, most current stream processing applications are more suitable for push-based processing, where data are pushed for processing as they become available at the data source. 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 appropriate event handler of the consumer. While the other two methods create heap-allocated event objects to be passed to and processed by the next stage, event handling passes information through calls to event handlers, which are frame-allocated. That is, event handling avoids the overhead of garbage collecting the event objects at the end of the pipeline. One example of XML event handling is SAX parsing [19]. It is easy to adapt a state transformer to be used as an event handler: the part of the code that analyzes an event type becomes an event handler for this type, while the code that creates and returns an event, becomes a call to the appropriate event handler of the consumer. Although our framework is independent of the streaming method used, our implementation is based on SAX.

In our framework, a state transformer is a black box that can only recognize and process regular stream events (without updates). This eases the task of developing code for the XQuery processor because the coding effort would be the same as that for handling streams without updates, which has already been addressed by earlier work (by us as well as by others). More specifically, our goal is to develop a universal wrapper for any kind of state transformation that handles update events while letting the state transformer handle the regular events. But how can we do this effectively without any knowledge about the state and state transitions? As we will see next, the only function a state transformer needs to provide to this wrapper is *state adjustment*, $\mathsf{adjust}(s_1, s_2, s_3)$ of type $\mathcal{S} \times \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$, which must be defined for each non-inert state transformer. The result of $\mathsf{adjust}(s_1, s_2, s_3)$, given that the state s_2 has been changed to s_3 by some earlier update, is the adjustment of the state s_1 that reflects this update. Basically, for any stream region open to updates that has already passed through a state transformer, the state transformer keeps a state (or state history). If a region is updated so that its state is replaced from s_2 to s_3 then all regions that have arrived later in the stream should replace their state from s_1 to $\mathsf{adjust}(s_1, s_2, s_3)$ (the details of when and how will be given in Section 8).

The adjust function must satisfy the following properties $\forall s_1, s_2, s_3 \in \mathcal{S}$ and $\forall v \in \mathcal{WF}_i$ with $\mathsf{stream}[i] = k$:

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

In particular, from the last two properties, we have:

$$\mathsf{adjust}(f_k^*(v, s_1), s_1, s_2) = f_k^*(v, s_2)$$

which indicates that, if s_1 were changed to s_2, then $f_k^*(v, s_1)$ is adjusted to $f_k^*(v, s_2)$, as expected. If there is such a function, the state transformer is called *adjustable*. In our implementation of XQuery, all state transformers turned out to be adjustable. For example, the state transformer that counts text at any depth needs the following adjustment:

$$s_1.\text{count} + (s_3.\text{count} - s_2.\text{count})$$

That is, if s_2.count changes to s_3.count due to some update, then the value of s_1.count after any subsequent update should be adjusted as shown. For an inert state transformer, we have $\forall s_1, s_2, s_3 : \mathsf{adjust}(s_1, s_2, s_3) = s_1$, that is, previous state transitions do not affect the state s_1.

7. TRANSLATION OF THE TEMPORAL EXTENSIONS

A temporal restriction in our extended XQuery syntax takes the form $e\#v$, $e?t$, $e\#[v]$, or $e?[t]$. It should only affect the replacement updates applied to the "top-level elements" in the stream output of e. Consider for example the query ./a#1/b#2 applied to the stream subsequence given in Section 3. The temporal restriction #1 should only apply to the sR(2,5)/eR(2,5) replacement update, which is over the a-tagged elements, which are the top-level elements of the stream. After the /b#2 step, the top-level elements are the b-tagged elements. At that point, the temporal restriction #2 should apply to sR(3,4)/eR(3,4) replacement updates only. Therefore, each replacement update must be associated with a temporal qualifier so that each succeeding operator in the query pipeline that handles this update would use the qualifier to update the states properly. This can be done effectively if we add the temporal qualifier as an extra parameter to the sR event. More specifically, the sR qualifier is the string "#v", "?t", "#[v]", or "?[t]". When the sR events are introduced for the first time, their qualifier is set to $\perp$, which is equivalent to #0 (snapshot mode).

Given an XQuery expression e with a temporal restriction q, we construct the query pipeline of e and we append to this pipeline a new pipeline component, $\mathcal{T}$, that implements the functionality of the temporal restriction q. The state of $\mathcal{T}$, s, is the nesting level, used in identifying the replacement updates over top-level elements. The state transformer of $\mathcal{T}$ is $f(e, s)$:

```
if e = sE
  then return ([ e ], s + 1)
  else if e = eE
     then return ([ e ], s − 1)
  else if e = sR and e.qualifier = ⊥ and s = 0
     then return ([ sR(e.uid,e.id,q) ], s)
  else return ([ e ], s)
```

that is, when the beginning of a replacement update is received at the zero nesting level and has not been assigned a temporal qualifier yet (ie, its qualifier is $\perp$), we pass it through with the new qualifier, q. All other events are passed through as is. Note that, the expression $q/quote - $q/quote#1 in Query 2 will clone the stream associated with the variable $q (ie, it will produce two identical virtual streams with different stream numbers and ids by repeating each event twice under different ids). That way the replacement updates in $q/quote and $q/quote#1 will have different temporal qualifiers.

8. UPDATING HISTORY LISTS

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. Recall that update histories are kept for a replacement update only, which may replace any mutable region, including another replaced region, thus forming a chain of updated versions. Since a replace update starts with the same state as the one being replaced but may yield a different state at the end of the update, each update region is associated with a single starting state and an ending history, which is a list of states. For an update region not associated with a replace update, the ending history contains exactly one state.

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 that are affected by this update. That is, if we have two update regions A and B so that A has been applied earlier than B and we replace A, then we need to adjust the states of B too based on the A update. All state adjustments are applied at the end of an update.

The *wrapper state transformer*, $\mathcal{W}_f$, associated with a state transformer f, handles all events using the state transformer f and its state adjustment function exclusively. The state of $\mathcal{W}_f$ contains the following components, for each id:

- start[id]: the starting state associated with id (of type S).
- end[id]: the current/ending history list associated with id (of type $\mathcal{S}^*$).
- qualifier[id]: the temporal qualifier of a replacement update ($\perp$ otherwise).
- timestamp[id]: the timestamps of the updates in end[id] (of type int*).
- order[id]: the relative order of the update, if it were applied eagerly.

The end[id] is a history list that has at least one state, so that end[id]_0 is the current state (version 0), end[id]_1 is the previous version, etc. The *effective state* of an id, effective(id), depends on the temporal qualifier and is calculated from the end[id] list:

- If the qualifier[id] is $\perp$, then effective(id) is end[id]_0.
- If the qualifier[id] is $\#v$, then if $v <$ size(end[id]) then effective(id) is end[id]_v, otherwise it is ().
- If the qualifier[id] is $?t$, then effective(id) is end[id]_i such that $\text{timestamp[id]}_i \leq \text{current\_time} - t < \text{timestamp[id]}_{i-1}$ (if state $i-1$ exists), otherwise it is ().
- If the qualifier[id] is $\#[v]$, then effective(id) is s_j, for $j = \min(v, \text{size(end[id])})$, where s_j is calculated from the inductive definition:

$$\begin{aligned} s_0 &= \text{end[id]}_0 \\ s_i &= \text{adjust}(\text{end[id]}_i, \text{start[id]}, s_{i-1}) \end{aligned}$$

  That is, we are sequencing all states end[id]_i, for $i \in [0, j]$, using state adjustment (as it would have been with repeated insert-after updates).
- If the qualifier[id] is $?[t]$, then effective(id) is s_j given by the previous inductive definition, for $\text{timestamp[id]}_j \leq \text{current\_time} - t$.

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

```
if e is a regular stream event
  then return f(e,effective(e.id))
else 'update the states';              // described below
    return ([ e ], s )
```

That is, for non-update events, the state transformer works directly on the effective state. The states are updated with the help of the function enqueue(id,s) defined as:

```
insert s into the beginning of the end[id] queue
insert current_time into the beginning of the timestamp[id]
if the qualifier[id] is #v or #[v]
  then remove entries end[id]_i and timestamp[id]_i with i > v
else if the qualifier[id] is ?t or ?[t]
      then remove expired entries end[id]_i and timestamp[id]_i
                  with timestamp[id]_i > current_time − t
```

That way history lists are minimal since all expired entries are truncated immediately. When an update event is received, the states are updated as follows, based on the type of the event:

sM(i,j), sA(i,j):	start[j] ← effective(i); end[j]$_0$ ← effective(i)
sB(i,j):	start[j] ← start[i]; end[j]$_0$ ← start[i]
sR(i,j,q):	qualifier[j] ← q; start[j] ← start[i]; enqueue(j,start[i])
eM(i,j):	end[i]$_0$ ← effective(j)
eA(i,j):	adj(j,start[j],effective(j))
eB(i,j), eR(i,j):	adj(j,effective(i),effective(j))

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

```
for each i : order[i] > order[j] do
    start[i] ← adjust(start[i],s1,s2)
    for each j: end[i]_j ← adjust(end[i]_j,s1,s2)
```

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

sM(i,j): order[j] ← order[i]
sR(i,j,q): 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.

9. THE RESULT DISPLAY

The last pipeline stage is the result display, which displays the query results continuously. Recall that every update in the input stream is propagated as is through every stage in the query pipeline, until it reaches the result display where it causes an update to the actual text content of the display. The display is an editable text window, where text can be inserted, deleted, and replaced at any position. It supports the following methods:

- remove(start_position,end_position): Remove the text between the two positions.
- insert(position,text): Insert the text at the position.

The state of the display state transformer, s, is an integer, which is a position in the result display. The display state transformer $f(e, s)$ is:

```
if e = sE
   then return ([ ], ins(e.id, s, "<e.tag>"))
else if e = eE
   then return ([ ], ins(e.id, s, "</e.tag>"))
else if e = cD
   then return ([ ], ins(e.id, s, "e.text"))
else return ([ ], s)
```

where ins is a function that inserts the text at some position and adjusts the positions of the succeeding update regions using adj, defined in Section 8:

```
ins(id,position,text) =
      insert(position,text);
      adj(id,position,position+size(text));
      return position+size(text)
```

Although this state transformer is not a pure function (since it updates the display and adjusts the states), it is nevertheless adjustable:

adjust(s_1,s_2,s_2) = $s_1 + s_3 - s_2$

but requires a special side effect for the sR(i,j,q) event:

```
remove(start[i],effective(i)-start[i]);
'all other sR state updates from W_f'
```

while all other update events are handled in the same way as in the wrapper state transformer $\mathcal{W}_f$.

10. RELATED WORK

The query processing model presented in this paper is based on our earlier work on unblocking XQuery stream processing (XFlux [7]). XFlux is an architecture for processing continuous queries over continuous update streams of XML data. Instead of eagerly performing the updates on cached portions of the stream, this architecture propagates 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. This framework was used 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, thus minimizing the buffering needs of these updates at the end of the pipeline. Based on this framework, we developed methods for unblocking a number of important blocking/unbounded stream operations in XQuery using a small memory footprint, including concatenation, general predicates, descendant and backward steps, and sorting. The work reported in this paper extends our previous work by adding a temporal dimension to our continuous query processing framework to extract histories of updates that occur within prespecified sliding windows and use them in answering temporal queries.

Our XQuery temporal annotations have been based on our earlier work on the filler-hole fragmentation model for XML data and the temporal query language XCQL [4]. The query processing framework for XCQL was based on a nested algebra suitable for streaming XML data, while the framework reported here is independent of the underlying processing model. In addition, all XCQL temporal annotations were compiled away to XQuery function calls by using the XQuery type checking system to lift the temporal types to sequences of versions and to compile the temporal annotations into FLWR iterations over these lists, which added another layer of complexity to query processing. These translations were not suitable for continuous updates because there was no straightforward way to shorten the lists. In our current work, we cache histories of states, rather than XML fragments, and these lists are truncated based on the temporal annotations. Furthermore, our current framework is integrated with our work on unblocking operations [7], thus providing an integrated solution to XML stream processing.

Our stream updates for XML and our temporal annotations for XQuery are related to the work on stored temporal XML data and queries. Multidimensional XML (MXML) [11] is an extension of XML that encodes the temporal dimensions and represents changes in an XML document. In MXML, the changes are always materialized as separate instances. τXQuery [10] has been proposed as a query language for temporal XML. Although the τXQuery language is based on XQuery, unlike our temporal extensions, it proposes basically two types of temporal modifiers, current and validtime, to denote current queries and sequenced queries. While the former slices the XML tree on the current snapshot, the latter derives sequences of validTime groups. There are many other efforts concentrating on sliding window stream processing for relational data, such as CQL [22], StreaQuel [5], COUGAR [3], and Aurora [1].

Our approach is motivated by the work on online aggregation and approximate data stream processing. Online aggregation [14] outputs partial aggregation results at regular intervals and reports progress with a confidence interval, which allows users to cancel queries when the result is "good enough". Approximate query processing over data streams are based on sliding windows [2] and sketches [6] to cal-

culate approximate answers to aggregations and joins using condensed synopses to summarize the state. In our work, instead of calculating approximate answers, we generate continuous updates that reflect the query results up to that point. Nevertheless, our method can be used in conjunction with these approaches to achieve even further state reduction. Our work is also related to incremental view maintenance where updates arrive at very high rates [18]. The granularity of our views is the states of the stream operators and our framework maintains these views in a uniform way.

11. CURRENT STATUS AND FUTURE WORK

Currently, we have implemented our stream processing framework for snapshot queries only. The system, called XFlux [7], can handle many essential XQuery features. We are planning to extend this framework with full temporal capabilities in the near future. The main assumption behind our method is that it is better to lazily propagate the updates through the query pipeline all the way to final result display and do state adjustments to incorporate the update effects. A high-performance streaming system, of course, must be able to choose 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). We are planning to investigate these optimizations in the near future.

12. REFERENCES

[1] D. J. Abadi, D. Carney, U. Cetintemel, et al. Aurora: A New Model and Architecture for Data Stream Management. In *VLDB Journal* 2003, (12)2:120–139.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *PODS'02*.

[3] P. Bonnet, J. Gehrke, and P. Seshadri. Towards Sensor Database Systems. In *Second International Conference on Mobile Data Management*, 2001.

[4] S. Bose and L. Fegaras. Data Stream Management for Historical XML Data. In *SIGMOD'04*.

[5] S. Chandrasekaran, et al. TelegraphCQ: Continuous Data flow Processing for an Uncertain World. In *Conference on Innovative Data System Research*, 2003.

[6] A. Dobra, M. N. Garofalakis, J. Gehrke, and R. Rastogi. Processing Complex Aggregate Queries Over Data Streams. In *SIGMOD'02*.

[7] L. Fegaras. Efficient Processing of XML Update Streams. Submitted to a conference. Available at `http://lambda.uta.edu/xflux.pdf`.

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

[9] L. Fegaras, R. Dash, and Y. Wang. A Fully Pipelined XQuery Processor. In *XIME-P'06*.

[10] D. Gao and R. T. Snodgrass. Temporal Slicing in the Evaluation of XML Queries. In *VLDB'03*.

[11] M. Gergatsoulis and Y. Stavrakas. Representing Changes in XML Documents Using Dimensions. In *XSym'03*.

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

[13] A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *SIGMOD'03*.

[14] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *SIGMOD'97*.

[15] R. Motwani, J. Widom, A. Arasu, et al. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *CIDR'03*.

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

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

[18] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay. How To Roll a Join: Asynchronous Incremental View Maintenance. In *SIGMOD'00*.

[19] SAX. At http://www.saxproject.org/.

[20] StAX: BEA's Streaming API for XML. At http://dev2dev.bea.com/xml/stax.html.

[21] XQuery 1.0: An XML Query Language. At http://www.w3.org/TR/xquery/.

[22] J. Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *DBPL'03*.