Data Stream Management for Historical XML Data

Sujoe Bose, Leonidas Fegaras

The University of Texas at Arlington
416 Yates Street, P.O. Box 19015
Arlington, TX 76019
{bose,fegaras}@cse.uta.edu

ABSTRACT

We are presenting a framework for continuous querying of time-varying streamed XML data. A continuous stream in our framework consists of a finite XML document followed by a continuous stream of updates. The unit of update is an XML fragment, which can relate to other fragments through system-generated unique IDs. The reconstruction of temporal data from continuous updates at a current time is never materialized and historical queries operate directly on the fragmented streams. We are incorporating temporal constructs to XQuery with minimal changes to the existing language structure to support continuous querying of time-varying streams of XML data. Our extensions use time projections to capture time-sliding windows, version control for tuple-based windows, and coincidence queries to synchronize events between streams. These XQuery extensions are compiled away to standard XQuery code and the resulting queries operate continuously over the existing fragmented streams.

1. INTRODUCTION

Many data management applications require the existence of time-varying data. For example, medical information systems store information on patient histories and how each patient responds to certain treatments over time. Financial databases use histories of stock prices to make investment decisions. Temporal databases provide a complete history of all changes to a database and include the times when changes occurred. This permits users to query the current state of the database, as well as past states, and even future states that are planned to occur.

There is a recent interest in a new type of data management based on streams of historical data. Stream data may be infinite or repeated and transmitted in a continuous stream, such as measurement or sensor data transmitted by a real-time monitoring system continuously. An infinite stream may consist of a finite data stream followed by an infinite number of updates. Since client queries may refer to past history as well as to current data, a data stream resembles a read-once temporal database where all versions of data must be available to the client upon request. For example, a server may broadcast stock quotes and a client may evaluate a continuous query on a wireless, mobile device that checks and warns on rapid changes in selected stock prices within a time period.

While temporal query languages were developed to handle stored historical data, continuous queries were designed to operated on streaming data. For example, sliding window queries capture streaming data in a moving fixed-size window and are executed in a continuous manner over each window [5, 3]. Streamed data is inherently associated with a valid or transaction time dimension that identifies the time instant when the data is valid or is generated. For example, a temperature sensor may indicate the temperature reading at a certain location taken at a particular time of the day. Streamed data can be broadly classified into two types: instantaneous or event based, which takes place at a certain point of time, and epoch or lifetime based, which is valid during a certain time interval. Instantaneous data is typically associated with sensor streams, such as traffic sensors, while lifetime data is associated with change-based or update streams, such as stock quotes.

Earlier work on continuous query processing has focused on relational data transmitted as a stream of tuples. Each tuple is typically associated with a timestamp, which allows the specification of time-sliding windows in continuous queries. For example, an update to a record may simply be a newly inserted record in the stream with the same key but with different timestamp and values. This makes the continuous queries over historical stream data easier to specify in relational form than in other forms. Recent proposals of continuous query languages, such as CQL [11, 15], extend SQL with simple temporal constructs, such as time-interval and version projections to specify time- and tuple-based windows, but do not support temporal joins to synchronize events between streams.

In our framework, data is transmitted in XML form, since it is now the language of choice for communication between cooperating systems. A typical configuration for our push-based data model consists of a small number of servers that transmit XML data over high-bandwidth streams and many clients that receive this data. Even though our model resembles a radio transmitter that broadcasts to numerous small radio receivers, our clients can tune-in to multiple servers at the same time to correlate their stream data. In contrast to servers, which can be unsophisticated, clients must have

SIGMOD 2004 June 13-18, 2004, Paris, France.
Copyright 2004 ACM 1-58113-859-8/04/06 . . . $5.00.

enough storage capacity and processing power to evaluate sophisticated, continuous queries on the XML data streams. Contrary to other data stream management systems, a client would have to register/unregister with a server only once using a pull-based web service, but would be capable of executing any number of continuous queries without registering the queries with the server. The server, on the other hand, would multicast its data to the registered clients without any feedback from them. Pushing data to multiple clients is highly desirable when a large number of similar queries are submitted to a server and the query results are large [7], such as requesting a region from a geographical database. Furthermore, by distributing processing to clients, we reduce the server workload while increasing its availability. On the other hand, a stream client does not acknowledge correct receipt of the transmitted data, which means that, in case of a noise burst, it cannot request the server to resubmit a data packet to correct the errors.

A data stream in our framework consists of a finite XML document followed by continuous updates to the document. It is essential, therefore, to specify XML updates unambiguously, that is, to uniquely identify the updated element in an XML document. Hierarchical key structures [10] can uniquely identify XML elements, such as elements with an ID attribute, but not all elements can be identified this way. Our approach to updating XML streams is novel in that it does not require keys to update XML elements. Instead, we fragment an XML document into manageable chunks of information. These chunks, or fragments, are related to each other and can be reassembled at the client side upon arrival. Our XML fragments follow the *Hole-Filler* model [12, 9] in which every fragment is treated as a “filler” and is associated with a unique ID. When a fragment needs to refer to another fragment in a parent-child relationship, it includes a “hole”, whose ID matches the filler ID of the referenced filler fragment. A document can be fragmented to produce fillers and holes in arbitrary ways and the fragments can be arranged to any depth and breadth in a document tree. A server may choose to disseminate XML fragments from multiple documents in the same stream, can repeat some fragments when they are critical or in high demand, can replace them when they change by sending delta changes, and can introduce new fragments or delete invalid ones.

The notion of the filler ID is similar to that of a surrogate key in relational databases or an OID in object-oriented databases, but the granularity of our data is a fragment, which may contain multiple elements, rather than a single XML element. In our Hole-Filler model, updating a fragment is simply streaming a new fragment with the same filler ID but with different timestamp and content. An insertion of a new child to a node is achieved by updating the fragment that contains the node with a new hole, and deletion of a child, by removing the hole corresponding to the deleted fragment. When a fragment is deleted all its children fragments become inaccessible. In essence, a fragment is the unit of updates: one can only replace the entire fragment, not parts of it. If an XML document is transmitted unfragmented, any update would have to transmit the entire updated document. It is essential, therefore, that a server does a reasonable fragmentation of data to accommodate future updates with minimal overhead.

At each instance of time, a client should be able to see the complete history of changes in the transmitted document since the beginning of time. The Hole-Filler model is too low a level for the client to execute high-level XQueries since it requires the user to navigate from holes to the corresponding fillers during querying. More importantly, since a hole may correspond to multiple fillers due to updates, historical queries would have to explicitly manipulate the timestamp component of fragments. Instead, we provide the client a virtual view of historical data upto the current time that is easy and intuitive to query. This temporal view is derived by merging all versions into one XML tree, by replacing each hole with the sequence of all fragments that correspond to the hole. This view is never materialized; instead XQueries over this view is translated into XQueries over the existing streams of fragments.

The rest of the paper is organized as follows: Section 2 proposes the XCQL language to perform continuous queries on a temporal view of stream data. Section 3 details the approach we adopt to process stream events and updates in a temporal context. Section 4 describes the streaming strategy employed to model events and updates as fragments of XML data embedded with contextual information. Section 5 covers the reconstruction of a temporal view of the transmitted fragments. Section 6 formalizes the translation of XCQL expressions to process fragments as and when they arrive, without waiting for complete materialization. Section 7 provides the results of our experimental evaluation, and Section 8 summarizes the contributions and proposes future work envisioned in this framework. Section 9 discusses the related work, and Section 10 concludes the paper.

2. THE XCQL LANGUAGE

Our query language, XCQL, is XQuery extended with temporal language constructs for effective querying of historical data. These extensions are compiled away when an XCQL query over the temporal view is translated into an XQuery over the fragmented stream. Instead of restricting the lifespan of input streams only, as is done in CQL [11], we let our temporal extensions apply to any XQuery expression.

Essential to our query language is the notion of time and time intervals. Time can be any XQuery expression of type xs:dateTime, which conforms to the ISO8601 extended format CCYY-MM-DDThh:mm:ss where CC represents the century, YY the year, MM the month and DD the day. The letter T is the date/time separator and hh, mm, ss represent hour, minute and second respectively [13]. The time duration used in expressions takes the form PnYnMnDTnHnMnS, where nY represents the number of years, nM the number of months, nD the number of days, T is the date/time separator, nH the number of hours, nM the number of minutes and nS the number of seconds [13]. In addition, time expressions may use the constant start to indicate the beginning of time and the constant now to indicate the current time, which changes during the evaluation of a continuous query.

The time interval, $[time_1, time_2]$, contains all the time points between and including $time_1$ and $time_2$. XCQL supports a number of syntactic constructs to compare two time intervals a and b, such as a **before** b, which is equivalent to a.t2<b.t3 when a=[t1,t2] and b=[t3,t4]. The time interval, $[time]$, contains just one time point: $time$, so is a shorthand for $[time, time]$. Our version numbering scheme uses integers $1, \ldots,$ last to number all versions of a fragment, starting from the first version, 1, and ending at the last version at the current time, last. A version interval, $[e_1, e_2]$, contains

all the versions $e_1, e_1+1, \ldots, e_2$, where e_1 and e_2 are integer expressions. As before, $[e]$ is a shorthand of $[e, e]$.

Time and version intervals can only be used in the following XCQL syntax:

e?time_interval	interval projection
e#version_interval	version projection
vtFrom(e)	interval begin
vtTo(e)	interval end

where e is any XCQL expression. Note that the interval and version projections may be used at the same time to restrict versions applicable within the time range. The conceptual temporal view of data is based on the assumption that any XML element has a lifespan (a time interval), which can be accessed using the vtFrom/vtTo functions. In reality, very few elements are typically directly associated with a lifespan. These are the elements that correspond to event or temporal fragments in the Hole-Filler model. The lifespan of any other element is the minimum lifespan that covers the lifespans of its children (or [start,now] for leafs). Any XQuery expression, even those that construct new XML elements from temporal elements, conforms with this temporal view since temporal information is propagated from children to parent. The default interval projection applied to an expression is [start,now], which has no effect on the temporal information of data. If the most recent data is desired, then e?[now] should be used. When an interval projection is applied to an expression, the lifespan of the resulting value will be equal to the intersection of the projection interval and the current lifespans of input (the semantics will be given in Section 6). On the other hand, only elements associated with event or temporal fragments have versions. When a version projection is applied to a temporal element (that is, an element associated with an event or temporal fragment), then the correct version is chosen (if exists) and its lifespan is used in a temporal projection over the children of the element. Even though the interval and version projections have very simple syntax, they are very powerful and can capture many operations proposed by others. For example, windowing, such as R[Partition by R.A Rows n] or R[Partition by R.A Range n Days] in CQL, can be simulated by performing the group by first using regular XQuery and applying a version or an interval projection respectively to the result.

The following are examples in XCQL:

1. Suppose that a network management system receives two streams from a backbone router for TCP connections: one for SYN packages and another for ACK packages that acknowledge the receipt. We want to identify the misbehaving packages that do not receive an acknowledgment within a minute:

   ```
   for $s in stream("gsyn")//packet
   where not (some $a in stream("ack")//packet
                         ?[vtFrom($s)+PT1M,now]
                 satisfies $s/id = $a/id
                    and $s/srcIP = $a/destIP
                    and $s/srcPort = $a/destPort)
   return <warning> { $s/id } </warning>
   ```

   where PT1M is one minute duration.

2. 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. This system can locate the position of a vehicle by joining the streams of two radars over both frequency and time and by using triangulation:

   ```
   for $r in stream("radar1")//event,
       $s in stream("radar2")//event
                ?[vtFrom($r)-PT1S,vtTo($r)+PT1S]
   where $r/frequency = $s/frequency
   return
      <position>
          { triangulate($r/angle,$s/angle) }
      </position>
   ```

   where 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.

3. Some vehicles, such as buses and ambulances, have vehicle-based sensors to report their vehicle ID and location periodically. Road-based sensors report their sensor ID 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, we would like to switch the light to green at a time that depends on the speed and the position of the ambulance:

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

3. OUR APPROACH

In our framework, we consider data from a wide variety of streaming data sources, such as computer applications that generate values automatically or through user interactions. The ability to synchronize between the streams by issuing coincidence queries, using simple extensions to the standard XQuery language, is one of the contributions of our work.

Events and updates generated by a data stream application are usually associated with context information. For example, a temperature sensor generating temperature readings is associated with the locational information of the sensor, which is primarily static. While events are dynamic, the event context information is primarily static, but changing at times. For example, a credit card account is associated with a credit limit, which changes from time to time. Charge transactions made on the card are associated with and are bounded by the credit limit on the account at that particular point in time. The charge transactions are the events, and the credit limit, along with other account information, forms a context for the event. We observe that when events are generated, the context information need not be sent every time, however, the events must be processed within the overall context in a historical timeline to accurately capture the semantics of the event within its global context. The context information may itself change at times, so we would like to capture its temporal extents in the same way as we do for regular events. Thus, we treat both events and updates as stream entities and identify them with their contextual locality so that they can be integrated together and processed uniformly in the historical timeline. Also, we are faced with a challenge of sending XML data in fragments, associated with a context such that the data can be processed within its context. We address this challenge by encoding fragments with a context id and using the id to construct a complete XML document, with the events as sequences with an associated valid time within their context, and updates as elements associated with temporal duration. The fragments encoded with structural context information and associated with temporal extents can be visualized as forming a complete XML document so that queries on the entire XML document view can be processed with XCQL expressions.

Our approach benefits from two aspects. First, since we only send data in fragments, which is a unit of information for both events and updates, the amount of data transfer is minimal and sufficient. Second, we can process temporal queries against these fragments in a continuous fashion as and when they arrive without waiting to materialize the fragments to a complete XML document. The data fragments in the stream, comprising of events and updates to event contexts, provide a temporal XML view in the historical timeline of the stream.

We illustrate our approach in Figure 1. The events stream generates an infinite sequence of fragments corresponding to real-life events. The update stream produces fragments corresponding to updates to the event context and other related temporal data. By capturing both events and updates as fragments with temporal extents, we provide a unified continuous query processing, using the XCQL language, to query both historical and current data.

While an event is associated with a single time point, updates are associated with a valid time duration with "from" and "to" extents, denoting the temporal duration of validity. However, for convenience and simplicity, we associate "from" and "to" extents to events also, both being the same as its valid time.

3.1 Running Example

As a running example consider a credit card processing system. The accounts present in the system are associated with temporal qualifiers indicating their lifetime. Moreover, accounts are associated with credit limits, which change from time to time. While charge transactions arrive in a continuous event stream, as charges gets posted to accounts, account-level updates, such as credit limit changes, arrive in an update stream.

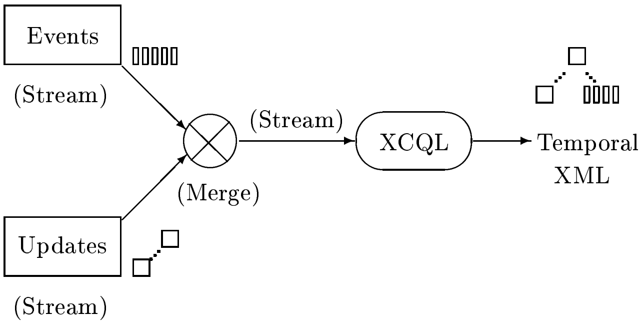

Figure 1: Event and Update Stream Processing

The DTD for the credit card system, with the associated temporal qualifiers, is the following:

```
<!DOCTYPE creditSystem [
<!ELEMENT creditAccounts (account*)>
<!ELEMENT account (customer, creditLimit*,
                   transaction*)>
   <!ATTLIST account id ID #REQUIRED>
   <!ATTLIST account vtFrom CDATA #REQUIRED>
   <!ATTLIST account vtTo CDATA #REQUIRED>
<!ELEMENT customer (#CDATA)>
<!ELEMENT creditLimit (#PCDATA)>
   <!ATTLIST creditLimit vtFrom CDATA #REQUIRED>
   <!ATTLIST creditLimit vtTo CDATA #REQUIRED>
<!ELEMENT transaction (vendor, status*, amount)>
   <!ATTLIST transaction vtFrom CDATA #REQUIRED>
   <!ATTLIST transaction vtTo CDATA #REQUIRED>
<!ELEMENT vendor (#PCDATA)>
<!ELEMENT status (#PCDATA)>
   <!ATTLIST status vtFrom CDATA #REQUIRED>
   <!ATTLIST status vtTo CDATA #REQUIRED>
<!ELEMENT amount (#PCDATA)> ]>
```

Even though transactions are event-based, being associated with a single valid time extent, we encode their temporal dimension with both "from" and "to" time points, equal to the valid time of the event, for simplicity and uniformity to facilitate the XCQL translation. As charge transactions are made against credit cards, the transaction fragments are transmitted in the stream. Moreover, the transactions in an account may be questioned, suspended, revoked, charged at different points in time based on customer/vendor feedback. The following is a fragment of the temporal credit card system XML data:

```
<creditAccounts>
  <accounts id="1234" vtFrom="1998-10-10T12:20:22"
                      vtTo="2003-11-10T09:30:45">
    <customer> John Smith </customer>
    <creditLimit vtFrom="1998-10-10T12:20:22"
                 vtTo="2001-04-23T23:11:08">
      2000 </creditLimit>
    <creditLimit vtFrom="2001-04-23T23:11:08"
                 vtTo="now">
      5000 </creditLimit>
    <transaction id="12345"
             vtFrom="2003-10-23T12:23:34"
               vtTo="2003-10-23T12:23:34">
      <vendor> Southlake Pizza </vendor>
      <amount> 38.20 </amount>
      <status vtFrom="2003-10-23T12:24:35"
                vtTo="now">
        charged
      </status>
    </transaction>
    ...
  </accounts>
  ...
</creditAccounts>
```

Based on the temporal XML view of the credit system, we would like to answer the following useful queries.

Query 1 : Retrieve all accounts and their current creditLimit, which are maxed-out in the billing period of November 2003. Assume billing periods for all accounts start on the 1st of every month.

```
for $a in stream("credit")//account
where sum($a/transaction?[2003-11-01,2003-12-01]
         [status = "charged"]/amount) >=
      $a/creditLimit?[now]
return
  <account>
  { attribute id {$a/@id},
    $a/customer,
    $a/creditLimit }
  </account>
```

Note that the window specification, in our XCQL language, is blended into a temporal duration specification in a seamless XQuery compatible format. The window condition acts as a filtering criterion to group together matching elements along with other filtering conditions. Also note that the "transaction" elements are inherently grouped by accounts so that the transactions pertaining to each account may be processed as a group within the window/filter criterion.

Query 2 : Retrieve all accounts that exhibit potential fraudulent behavior, that is, to detect the credit cards that may be lost and misused, so that the account holder may be contacted and alerted. We need to determine all accounts whose transactions within an hour totals a value of more than a maximum of $5000 and 90% of its current credit limit.

```
for $a in stream("credit")//account
where sum($a/transaction?[now-PT1H,now]
         [status = "charged"]/amount) >=
      max($a/creditLimit?[now] * 0.9, 5000)
return
    <alert>
      <account id={$a/@id}>
        {$a/customer}
      </account>
    </alert>
```

4. FRAGMENTED XML DATA

As and when events are generated in the system, eg. when transactions are made against credit cards, the system will transmit the event data framents consisting of transaction information. Transmitting data in fragments, instead of sending the entire XML document, is expedient as only a part of the data, in our case transactions, needs to be transmitted. As and when new events get posted, a fragment comprising of the event data is transmitted along with information regarding the structural location of the event within its context. Similarly, for stream updates, only the fragment corresponding to the change has to be transmitted as none of the data in its structural context changes. Moreover transmitting in fragments is beneficial, as small fragments can be processed as and when they occur, as will be seen in the later sections. This increases the throughput of query processing, as fragments are processed as soon as they arrive without waiting to reconstruct the entire XML document before processing begins.

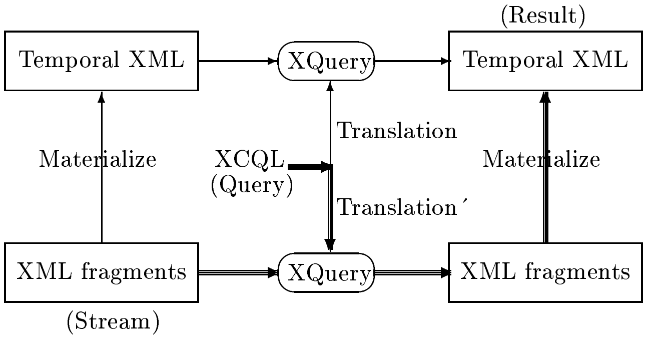

Figure 2: XCQL Query on Fragmented Stream

Our approach in transmitting XML fragments as and when they are generated, and processing XML fragments as and when they arrive, without reconstructing the complete XML document in its historical entirety, is summarized in Figure 2. The unit of transfer in our system is an XML fragment, which can relate to an event or an update to an XML element. When fragments are received, one simple option is to materialize the complete XML document based on the context information in the fragments. A query written in XCQL, is then compiled-out to translate interval and version projections into a standard XQuery, which is processed against the materialized XML. As an efficient alternative, instead of materializing a complete XML document, we would like to process the XML fragments as and when they arrive in a continuous fashion. Thus the XCQL query translation must now be cognizant of the context of a fragment in order to process the fragments prior to materialization. We transform the XCQL expressions into XQuery expressions to operate on fragments and then we materialize the resulting XML by reconstructing the processed result fragments with its context information.

4.1 Tag Structure

Our framework makes use of a structural summary of XML data, called the *Tag Structure*, which contains, along with structure of data, information about fragmentation and the temporal dimensions of data. The Tag Structure defines the structural make-up of the XML data stream, and captures all the valid paths in the data fragments. This information is used in the transformation process of XCQL expressions to operate on the XML fragments and during materialization of the result. Moreover, the Tag Structure is used while expanding wild-card path selections in the queries. Also, this structure gives us the convenience of abbreviating the tag names with IDs (not used here) for compressing stream data. The Tag Structure is structurally a valid XML fragment that conforms to the following simple recursive DTD:

```
<!DOCTYPE tagStructure [
<!ELEMENT tag (tag*)>
   <!ATTLIST tag type (snapshot | temporal | event)
                            #REQUIRED>
   <!ATTLIST tag id CDATA   #REQUIRED>
   <!ATTLIST tag name CDATA #REQUIRED> ]>
```

A tag is qualified by an id and name along with the type of the elements occurring with that tag name. Since there are 3 types of fragments that are possible in a streaming context, namely snapshot, temporal and event, we annotate the tag structure with the type of fragment to aid in query processing on the fragments. While a snapshot tag defines fragments as regular non-temporal elements that have no temporal dimension, the temporal tag defines fragments that have a "valid time from" and "valid time to" time dimensions, and the event tag defines fragments as transaction elements with only a "valid time" time dimension. The XML data is fragmented only on tags that are defined as temporal and event nodes only. The tag structure corresponding to the credit card application can be defined as follows:

```
<stream:structure>
 <tag type="snapshot" id="1" name="creditAccounts">
  <tag type="temporal" id="2" name="account">
   <tag type="snapshot" id="3" name="customer"/>
   <tag type="temporal" id="4" name="creditLimit"/>
   <tag type="event" id="5" name="transaction">
    <tag type="event" id="6" name="vendor"/>
    <tag type="temporal" id="7" name="status"/>
    <tag type="snapshot" id="8" name="amount"/>
   </tag> </tag> </tag>
</stream:structure>
```

The tag structure provides information on the location of holes in the filler fragments. When the tag type in the tag structure is "event" or "temporal", the corresponding tag in the stream will occur in a filler fragment. A tag annotated as "snapshot" will always be embedded within another temporal/event element or will be the root element, which is always static.

4.2 XML Fragments

The XML fragments are annotated with the Tag Structure Id (tsid) in order to readily infer its structural location, given the Tag Structure of the stream. Also, since fragment must identify the holes they fill in their context fragment, they are associated with filler ids. The filler and hole ids in the fragments form the link to reconstruct the entire XML document from the fragments. Moreover, the fragment is qualified by a validTime extent, denoting the time of its generation. As we will see in Section 5, the validTime in the filler container is used to derive the temporal dimension of the fragment contained in the materialized temporal view of the XML stream. In our credit system example, since the streamed charge transactions on credit cards must be associated to their accounts in the temporal stream view, we encode the transactions as fragments qualified with a filler id matching a hole in its account fragment along with its time of occurrence. The transmitted fillers fill holes in their contextual XML document, thus producing a temporal view of the entire stream. For example, the following filler transaction fragments will be transmitted as and when the charge transactions are made against a credit card.

filler 1:

```
<filler id="100" tsid="5"
                validTime="2003-10-23T12:23:34">
  <transaction id="12345">
    <vendor> Southlake Pizza </vendor>
    <amount> $38.20 </amount>
    <hole id="200" tsid="7"/>
  </transaction>
</filler>
```

filler 2:

```
<filler id="200" tsid="7"
                validTime="2003-10-23T12:23:35">
  <status> charged </status>
</filler>
```

From the example above, the "charge status" filler fragment 2, with $id = 200$, fills the hole, with $id = 200$, in its context "transaction" fragment. It can be noted that the fragmentation granularity used in the example above closely models the real-life behavior of requesting for a charge transaction and receiving a response of "charged" or "denied" at a later time. When the card is successfully charged, the entire "transaction" fragment need not be re-transmitted again, but only information relevant to the status, such as the confirmation number. The event generator, however, retains the knowledge of the fragments so that the appropriate hole/filler ids may be associated with the fragments. When the holes in above fragments are resolved, the materialized view will resemble the transaction element in the XML excerpt in Section 3.1.

Since the validity of a charge transaction on a credit card may be questioned by the account holder, the status of a transaction is associated with temporal extents to capture the time points at which the status changed as is illustrated below:

filler 3:

```
<filler id="300" tsid="5"
                validTime="2003-09-10T14:30:12">
  <transaction id="23456">
    <vendor> ResAris Contaceu </vendor>
    <amount> $1200 </amount>
    <hole id="400" tsid="7"/>
  </transaction>
</filler>
```

filler 4:

```
<filler id="400" tsid="7"
                validTime="2003-09-10T14:30:13">
  <status> charged </status>
</filler>
```

filler 5:

```
<filler id="400" tsid="7"
                validTime="2003-11-1T10:12:56">
  <status> suspended </status>
</filler>
```

Filler 3 and 4 correspond to the charge transaction performed on the card, and Filler 5 corresponds to a suspension of the charge caused by a investigation request by the customer at a later date. After Filler 5 is transmitted, any query that filters transactions for charges more than $1000, for example, must not report this transaction, as its status has now changed to "suspended". We will see how this is accomplished in the later sections.

5. RECONSTRUCTION OF THE TEMPORAL VIEW

The fragments received by a client may be reconciled for holes and the resulting temporal view materialized to form a complete XML document. Since fragments represent events and updates to the XML data with temporal extents, after materialization, elements in the resultant XML will be encoded with their temporal duration of validity. However, materializing the fragments to form a temporal XML document before processing is not our goal. We could do better by processing the fragments as is, and then materialize the result XML. The following function materializes an element of a fragment by replacing holes with fillers:

```
define function temporalize($tag as element()*)
  as element()*
{ for $e in $tag/*
  return if(not(empty($e/*)))
         then element {name($e)}
              {$e/@*, temporalize($e)}
         else if(name($e)="hole")
              then temporalize(get_fillers($e/@id))
         else $e }
```

Central to the temporalize method is the get_fillers function, which returns the filler fragments corresponding to a hole id along with their deduced temporal extents. The get_fillers method is used while traversing a hole to a filler based on the hole-id and is defined as follows:

```
define function get_fillers($fid as xs:integer)
  as element()
{ <filler id="{$fid}">
 { let $fillers := doc("fragments.xml")/fragments/
        filler[@id=$fid]
   for $f at $p in $fillers
   let $e := $f/* order by $f/@validTime
   return
    element {name($e)}
    {$e/@*,
     attribute vtFrom {$f/@validTime},
     attribute vtTo
       { if ($p = count($fillers))
         then "now"
         else $fillers[$p+1]/@validTime },
     $e/node()} }
  </filler> }
```

The get_fillers method encases the versions of elements present in matching filler ids into a filler with the same id. This is required so that we can perform path projections to extract the required elements from the fillers, since a context element may contain holes pertaining to different elements. For example, the "account" fragment has holes for both the "creditLimit" (temporal) and the "transaction" (event) subelements. So the get_fillers method on the transaction may yield fillers with both "creditLimit" and "transaction" elements. By performing the "creditLimit" path projection, we will get only "creditLimit" elements, and "transaction" path projection, "transaction" elements.

The temporalize method replaces all occurrences of "holes" with fillers based on the hole id. Thus the materialized result is a temporal XML, devoid of holes. As we mentioned before, we will materialize the result only when a query has completed execution and the results ready to be rendered. Notice that the temporalize function is a recursive one, since holes could be present anywhere deep in the filler chain. In the following section, we will see how we can flatten this recursive function by leveraging schema information on the stream.

5.1 Schema-Driven Reconstruction

To remove recursion from the get_fillers function that temporalizes a fragmented XML stream, we leverage the schema of the stream in the form of a Tag Structure presented in Section 4.1. Since the tag structure provides information on the location of holes in the fragments and the type of fragments, the following instance of temporalize is generated automatically from the Tag Structure:

```
define function
temporalizeCreditAccounts($e1 as element())
  as element()
{ <creditAccounts>
  { for $e2 in get_fillers_list(
        $e1/creditAccounts/hole/@id)/account
    return
      <account>
      {$e2/@*,
       $e2/customer,
       for $e3 in get_fillers_list($e2/hole/@id)/*
       return
       if(name($e3) = "creditLimit") then $e3
       else
           <transaction>
           { $e3/@*,
             $e3/vendor,
             $e3/amount,
             for $e4 in get_fillers_list(
                 $e3/hole/@id)/status
             return $e4 }
           </transaction> }
      </account>}
   </creditAccounts>}
```

The tags that correspond to a type "event" or "temporal" in the Tag Structure will appear as separate filler fragments. Thus, while elements with tag type "snapshot" are accessed with a direct path projection, those qualified as "event" and "temporal" are accessed by resolving the holes, using the get_filler function. When we come across a tag with type "event" or "temporal", we use the hole-id in the fragment to get all the fillers corresponding to the hole. Since the get_fillers function only returns the various versions of a single element corresponding to a hole id, we use a list-variant of the get_fillers function, get_fillers_list, to return a set of versions of elements corresponding to a set of hole ids, since a filler could have several holes relating to its various child elements. The get_fillers_list function is defined as follows:

```
define function
get_fillers_list($fids as xs:integer*) as element()*
{ for $fid in $fids
  return get_fillers($fid) }
```

After the fillers are retrieved based on the hole ids, we perform path projections to retrieve the elements based on the tag structure, and the fillers are further explored for embedded holes. For example, the "accounts" tag could have elements, which are either "creditLimit" or "transaction". In the former case, we do not need to navigate further, since a "creditLimit" element does not have other child elements, whereas the "transaction" element has other sub-elements, which need to be explored recursively.

6. XCQL TRANSLATION

Since XCQL is XQuery with temporal extensions, our first step would be to translate XCQL expressions to XQuery expressions so that the queries in our framework can be evaluated using a XQuery compliant query processor. The fragments we receive with temporal extents can be materialized to produce a temporal XML document on which the translated query will be processed. However, since we would like to process the fragments *ad verbatim* without materialization, the XCQL expressions would need to handle the presence of holes within fillers in order to be able to continue processing over the filler fragments having holes. We observe that only the path traversal expressions of an XCQL expression need to be translated to handle holes in filler fragments when a path expression crosses-over a hole. Thus the following path traversal components of an XCQL (XQuery) expression, e, are to be translated to handle holes.

e	$::=$	$stream(x)$	stream accessor
	$\vert$	e/A	path projection
	$\vert$	$e/*$	any projection
	$\vert$	$e//A$	wild-card path projection
	$\vert$	$e/@A$	attribute projection
	$\vert$	$e[pred]$	predicate
	$\vert$	$e?[tb, te]$	interval projection
	$\vert$	$e\#[vb, ve]$	version projection

We employ the tag structure to facilitate the path translation to handle holes. Based on the tag structure, if an element is qualified as a "temporal" or "event" type, then we know that the element will be present only as a filler fragment filling a hole in the context element. We use the get_fillers function to find all the fillers having ids matching the hole id. The mapping function that converts the XCQL path expressions into a valid XQuery is presented using denotational semantics in Figure 3.

In Figure 3, $e : ts \rightarrow e'$ indicates that e has tag structure ts and is translated into e'. When a path projection is encountered, the tag type in the Tag Structure is used to determine if a hole resolution must be performed. If the tag type is "snapshot" then the element will be directly embedded in the context, or will be present in another filler otherwise. In the latter case, we use the hole id in the context element to find out the fillers filling this hole. The get_filler method is used to determine the versions of the filler elements that fill the hole. The new tag structure of the projection is derived by going one level deeper into the tag tree using the projection tag name.

For a wild-card tag projection, since the tag structure describes all possible paths in the fragments, we traverse through all child elements recursively, each time going one level deeper into the tag structure, until we reach the termination condition of the presence of the tag. For an attribute projection, since we do not have version attributes, the mapping is straightforward.

Since a filtering condition can also be a XCQL expression, we interpret the *pred* in the path expression in the current context before applying the predicate to its context path. We translate the interval projection and the version projection into XQuery functions to keep the translation modularized. The interval projection finds the set of elements having a valid temporal duration within the duration specified in the filter. It performs temporal slicing on the elements, by recursively navigating into the fragment, traversing "holes" in the process, so that the temporal extents of all the elements in the sub-tree fall within the specified range. The interval_projection method is defined as follows:

```
define function interval_projection1($e as element(),
  $tb as xs:time, $te as xs:time)
{ if(empty($e)) then ()
  else if(name($e) = "hole") then
    for $f in get_fillers($e/@id)
    return interval_projection($f,$tb,$te)
  else if (empty($e/@vtFrom)) then
    element {name($e)}
    { if(empty($e/*)) then $e/text()
      else for $c in $e/*
      return interval_projection($c,$tb,$te)}
  else if($e/@vtTo lt $tb or $e/@vtFrom gt $te)
      then ()
  else
    element {name($e)}
    { attribute vtFrom {max($e/@vtFrom,$tb)},
      attribute vtTo {min($e/@vtTo,$te)},
      if(empty($e/*)) then $e/text()
      else for $c in $e/*
      return interval_projection($c,$tb,$te) }
}
define function interval_projection($e as element()*,
$tb as xs:time, $te as xs:time) as element()*
{ for $l in $e
  return interval_projection1($l, $tb, $te) }
```

In the above function, it is assumed that $\$tb \leq \$te$. Also notice that the temporal extents of elements, which intersect with the input range, are clipped to fall within the range of intersection. When there is no interval projection, it is implied e?[start,now], that is, for elements other than snapshot, all the versions of the context elements are returned in the system. The version projection on the other hand uses the index of the elements in their historical timeframe to determine the elements, whose index falls within the version range requested. After the version elements are selected, the time interval of those elements are used to determine the temporal extents of the child elements in the tree that needs to be retrieved. Thus the version_projection method determines the right version(s), and then uses the interval_projection method to perform a temporal slicing on the subtree. The version_projection method is defined as follows:

$\text{stream}(x) :\ \text{TagStructure}(x) \to \text{get\_fillers}_x(0)$

$e/* :\ (ts_1, \ldots, ts_n) \to (e_1, \ldots, e_n)$ where $\forall c_i \in ts/tag/@name :\ e/c_i :\ ts_i \to e_i$

$e//A :\ (ts',\, ts_1, \ldots, ts_n) \to (e',\, e_1, \ldots, e_n)$ where $e/A :\ ts' \to e'$ and $\forall c_i \in ts/tag/@name :\ e/c_i//A :\ ts_i \to e_i$

Given that $e :\ ts \to e'$, then :

$e/@A :\ ts \to e'/@A$

$e/A :\ ts/\text{tag}[@\text{name} = \text{“}A\text{”}] \to e'/A$ if $ts/\text{tag}[@\text{name} = \text{“}A\text{”}]/@\text{type} = \text{“snapshot”}$

$e/A :\ ts/\text{tag}[@\text{name} = \text{“}A\text{”}] \to \text{get\_fillers}_x(e'/\text{hole}/@id)/A$ otherwise

$e[pred] :\ ts \to e'[pred']$ where $pred :\ ts \to pred'$

$e?[tb, te] :\ ts \to \text{interval\_projection}(e', tb, te)$

$e\#[vb, ve] :\ ts \to \text{version\_projection}(e', vb, ve)$

Figure 3: Schema-based translation ($e :\ ts \to e'$ means that e has tag structure ts and is translated into e')

```
define function version_projection($e as
element()*, $vb as xs:integer, $ve as xs:integer)
as element()*
{ for $item at $pos in $e
  where $pos >= $vb and $pos <= $ve
  return
    element {name($item)}
    { $e/@*,
      for $c in $e/*
      return
      interval_projection($c,$e/vtFrom,$e/vtTo)
    } }
```

In the version_projection method, we assume $\$vb \leq \$ve$. As an example of version_projection, the following XCQL expression:

```
stream("credit")
  //transactions[vendor="ABC Inc"]#[1,10]
```

will retrieve the first ten transactions by vendor “ABC Inc” recorded in the system. Note that the application of the version projection has the same semantics as specifying a tuple window with a grouping operation. Our XCQL language version/interval expressions thus blend naturally with an XQuery predicate filter. When the input element is a “snapshot”, the version_projection considers it as a single version element. Note that holes embedded within the context element sub-tree in a version_projection, are resolved by the interval_projection function.

6.1 Example Translations

Consider the following simple XQuery to return all transactions that carry a charge amount of >$1000.

```
for $t in stream("creditSystem")/creditAccounts/
  //transaction
where $t/amount > 1000 and $t/status = "charged"
return $t
```

The query is transformed to interrogate filler fragments as follows:

```
for $t in get_fillers(
           get_fillers(
            get_fillers(0)/creditAccounts/hole/@id
           )/account/hole/@id
          )/transaction
    where $t/amount > 1000 and
        get_fillers($t/hole/@id)/status = "charged"
    return $t
```

The “creditAccounts” element is defined as a snapshot type, and hence does not need hole traversals. However, since the “account” tag is defined as “temporal” type, the get_filler method is used to traverse the holes to get all accounts. The same holds for the “transaction” and the “status” elements. Note that due to the existential semantics of XQuery path expressions, the above query will retrieve *filler 3* in Section 4.2, since it requires at least one “status” element to be “charged”. The more accurate way of writing this query would be:

```
for $t in stream("creditSystem")/creditAccounts/
  //transaction
where $t/amount > 1000 and
      $t/status?[now] = "charged"
return $t
```

In this case, the query is transformed with an interval projection as follows:

```
let $now := currentDateTime()
for $t in get_fillers(
           get_fillers(
            get_fillers(0)/creditAccounts/hole/@id
           )account/hole/@id
          )/transaction
    where $t/amount > 1000 and
      interval_projection(
        get_fillers($t/hole/@id)/status, $now, $now)
            = "charged"
    return $t
```

The above query would not retrieve the *filler 3*, since its current status, after *filler 5* is received, is “suspended”. Note that, in this example, we could have also used e#[last] to achieve the same result.

The translation for *Query 1* in Section 3.1, which determines the accounts that have maxed-out in the month, is as follows:

```
let $now := currentDateTime()
for $a in
  get_fillers(
    get_fillers(0)/creditAccounts/hole/@id
  )/account
```

```
where
  sum(interval_projection(
     get_fillers($a/hole/@id)/transaction,
     "2003-11-01","2003-12-01")
     [get_fillers(./hole/@id)/status= "charged"]
     /amount) >=
   interval_projection(
     get_fillers($a/hole/@id)/creditLimit,
     $now,$now)
return
  <account>
  { attribute id {$a/@id},
      $account/customer,
       get_fillers($a/hole/@id)/creditLimit }
  </account>
```

Any “transaction” XML fragment occurring in the stream as an event, falling within the interval_projection of the month duration, is filtered and is used to compute the cumulative charge amount for the month. This is then compared with the current creditLimit to determine if the card has exceeded its limit. If so, further transaction requests are denied.

Query 2 in Section 3.1, is translated as follows:

```
let $now := currentDateTime()
for $a in
  get_fillers(
    get_fillers(0)/creditAccounts/hole/@id
  )/account
where
  sum(interval_projection(
     get_fillers($a/hole/@id)/transaction,
     $now-PT1H,$now)
     [get_fillers(./hole/@id)/status= "charged"]
     /amount) >=
   max(interval_projection(get_fillers($a/hole/@id)
      /creditLimit, $now, $now)
       * 0.9, 5000)
return
    <alert>
      <account id={$a/@id}>
        {$a/customer}
      </account>
    </alert>
```

The temporal duration qualifiers in the above translation, eg. PT1H, are actually embedded in a xdt:dayTimeDuration XML Type constructor in our implementation, but not shown here. Thus the temporal queries defined using our XCQL language are translated to process XML fragments directly instead of materializing the temporal view. Note that the window specification in our framework is blended into a temporal query construct to provide clear semantics of stream query processing. By treating both stream events and updates to XML data in our framework, we provide a unified query language and processing methodology well suited for efficient XML data exchange.

7. EXPERIMENTAL EVALUATION

We have implemented the XCQL translator in Java, which translates XCQL queries into XQueries operating directly on fragments of XML data, as outlined in Section 6. We have used the Qizx XQuery Processor [22] to process the translated queries and we have evaluated our filler-processing query translations against the XMark [23] benchmark framework. Our query processing approach on fragmented XML streams is to process the queries directly on filler fragments before reconstructing the result, as opposed to reconstructing the complete document by reconciling the holes, and then executing the query on the materialized document. We have selected three representative queries, Q1, Q2 and Q5, from the XMark framework to compare the performance on selective, range, and cumulative queries. The experiments were run on datasets generated by the xmlgen program to produce an auction XML stream, provided by the XMark framework, with various scaling factors of 0.0, 0.05 and 0.1. We have written an XML fragmenter that fragments an XML document into filler fragments, based on the tag structure defining the fragmentation layout. The first method of query execution is to construct the entire document from the filler fragments and then executing the query on the document (CaQ). The second method is to process the fragments directly, thereby filtering those fillers that would not be part of the result set and then constructing the resulting document (QaC). The third method leverages the tag structure id encoded in the filler fragments to process only those fillers that are required by the query (QaC$^+$). While in QaC, the fillers are resolved starting from the root fragment, on demand, along the query execution path, resolving holes along the path traversal. In (QaC$^+$), only the fillers required by the query are processed without resolving the holes in other levels not required by the query. We ran the experiments on a 1.2Ghz Intel Pentium III processor, with 512MB RAM, under normal load. The results of the experiments are summarized in Figure 4.

To illustrate the differences in the execution of the query methods on the filler fragments, consider the aggregate query Q5 that determines the number of auctions that have closed with a price of over $40, defined in the XMark framework as:

```
count(for $i in document("auction.xml")/site/
                closed_auctions/closed_auction
        where  $i/price/text() >= 40
        return $i/price)
```

In the CaQ method, this query is translated to first materialize the entire document and then to execute the query as follows:

```
count(for $i in temporalize(get_fillers(0))/
                closed_auctions/closed_auction
        where  $i/price/text() >= 40
        return $i/price)
```

In the QaC method, the query is translated to operate on the fragments, starting from the root, recursively reconciling the holes, as follows:

```
count(for $i in get_fillers(
                  get_fillers(0)
                    /site/closed_auctions/hole/@id
                )/closed_auction
        where  $i/price/text() >= 40
        return $i/price)
```

In the QaC$^+$ method, the query is translated to operate only on the fragments required by the query path, guided by the tag structure-based mapping of the path to the tsid, as follows:

Query	File Size	Fragmented File Size	Method	Run Time
Q1	27.3Kb	34.8Kb	QaC$^+$	161ms
			QaC	190ms
			CaQ	320ms
	5.8Mb	6.9Mb	QaC$^+$	1,723ms
			QaC	49,391ms
			CaQ	335,843ms
	11.8Mb	13.9Mb	QaC$^+$	3,966ms
			QaC	197,354ms
			CaQ	1,799,207ms
Q2	27.3Kb	34.8Kb	QaC$^+$	190ms
			QaC	200ms
			CaQ	341ms
	5.8Mb	6.9Mb	QaC$^+$	4,487ms
			QaC	45,385ms
			CaQ	353,248ms
	11.8Mb	13.9Mb	QaC$^+$	8,222ms
			QaC	199,016ms
			CaQ	1,859,073ms
Q5	27.3Kb	34.8Kb	QaC$^+$	160ms
			QaC	201ms
			CaQ	310ms
	5.8Mb	6.9Mb	QaC$^+$	1,763ms
			QaC	19,528ms
			CaQ	335,382ms
	11.8Mb	13.9Mb	QaC$^+$	3,095ms
			QaC	110,409ms
			CaQ	1,886,022ms

Figure 4: Experimental Results

```
count(for $i in doc("auction_fillers.xml")/fragments
                  /filler[@tsid=603]/closed_auction
        where  $i/price/text() >= 40
        return $i/price)
```

where *tsid* = 603 matches the tag id for the closed_auction filler fragments in the fragmented auction XML document.

From the experimental results we observe that at higher loads, the QaC method outperforms the CaQ method by an order of magnitude, and is outperformed by the QaC$^+$ method by an order of magnitude. The difference in performance widens in queries such as query Q1 and Q5, as the queries are more selective, and the lesser performing methods spend much time reconciling the holes, which would otherwise not be needed in the query. In the QaC$^+$ method, however, the fillers are retrieved using the tsid attribute and the path expression does not require any hole reconciliation.

8. SUMMARY AND FUTURE WORK

In our framework, we propose XCQL as a XQuery extension to process streaming XML data in a temporal dimension. The queries operate seamlessly on fragments without materializing the entire stream to execute the temporal query. Moreover, XML-encoded stream events and XML data updates are captured with temporal extents and the XCQL query is performed with consistent semantics. We prove the feasibility of our proposed approach of processing directly the fragments prior to re-construction instead of materializing the document and then executing queries, by means of experimental evaluation using the XMark benchmark. Although we have addressed the processing of XML fragments directly without materialization, we have not addressed the mechanics of scheduling the fragments through the XCQL query tree. Techniques for Operator scheduling in a stream management system have been proposed in [20, 18] and will be addressed in our framework as future work. We have not addressed versioning attributes in our framework. Although we can accommodate attribute versioning in our existing framework by versioning the elements having the attributes, we will address this comprehensively as part of future work. τXQuery [2] has handled attribute versioning by constructing pseudo-elements to capture the time extents of temporal element attributes. Also, we would like to consider the effects of ID/IDREFs on temporality of the stream data and the handling of recursive XML, which are not currently supported in our current framework. Since our translation relies heavily on efficiency of the get_fillers function, we would like to research optimization techniques to unnest/fold the get_fillers functions using language rewriting rules. An alternative visualization of the get_fillers method would be a join between the hole-ids and the filler-ids so that various join optimizations may be employed.

9. RELATED WORK

The Tribeca [4] data stream processing system provides language constructs to perform aggregation operations, multiplexing and window constructs to perform stream synchronization, however, is restricted to relational data. Other efforts concentrating on windowed stream processing, such as CQL [11], StreaQuel [16], COUGAR [1], AQuery [17] also addresses only relational data, and provides SQL-like constructs to process streaming data. In our framework, we address streams of XML data, which is inherently hierarchical and semistructured. We have developed XCQL as a simple extension to the XQuery language to perform complex stream synchronization, grouping and aggregation. Moreover, since our language is based on XQuery, it provides seamless integration between streamed and stored XML data sources without additional conversion constructs. Moreover, we have envisioned a unified model for processing events and updates to XML data as fragments, wherein data may be transmitted in manageable chunks closely modeling real-life behavior. The COUGAR [1] system proposes the use of ADTs in object-relational database systems to model streams with associated functions to perform operations on the stream data. The Aurora system [18, 14] provides a graphical interface tool to build operator trees that will be processed as the data streams through, and addresses operator scheduling to improve system efficiency of the continuous queries.

Temporal coalescing [6] addresses the merging of temporal extents of value-equivalent tuples. In our framework, temporal coalescing is performed by capturing the time of occurrence of a change in a filler fragment. When a query is executed, fillers are interrogated in the order of their validTime timestamp and the temporal extents of the fragment are determined before the query is evaluated. In [8], Multidimensional XML (MXML), an extension of XML is used to encode the temporal dimensions and represent changes in an XML document. Moreover, in MXML, the changes are always materialized as separate instances. A variant of the hole-filler model, for navigating XML data, has been proposed in [19], however, in the context of pull-based content navigation over mediated views of XML data from disparate data sources. In our framework, the hole-filler model is used in a push-based streaming model, for fragmenting XML data to be sent to clients for continuous query processing in a historical timeline.

In [21], an XML-XSL infrastructure is proposed to encode temporal information into Web documents, and support temporal predicate specification. τXQuery [2] has been proposed as a query language for temporal XML. Although the τXQuery language is based on XQuery, unlike our XCQL language, 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 valid-Time groups. Compared to τXQuery, XCQL blends naturally to the XQuery language, with only minor extensions to perform powerful temporal and continuous stream synchronization. Also, we have shown how XCQL can be operated on streaming XML data, which arrives as fragments. Moreover, we provide a unified methodology to continuously process XML-encoded stream events and updates to XML data in an integrated historical timeline.

10. CONCLUSION

Our framework for data stream management is different from other proposals in terms of data model, query language, and query processing. Our data model of continuous XML updates and events, does not require the introduction of keys and the fragmentation necessary for updating is hidden from users. Our query language works on multiple XML data streams and is able to correlate and synchronize their data. Even though the virtual view of the historical streamed data is purely temporal, this view is never materialized; instead temporal queries are translated into continuous queries that operate directly over the fragmented input streams and produce a continuous output stream.

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

11. REFERENCES

[1] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proceedings of the Second International Conference on Mobile Data Management 2001*, pages 3–14.

[2] D. Gao, R. T. Snodgrass. Temporal Slicing in the Evaluation of XML Queries. In *VLDB 2003*, pages 632–643.

[3] L. Golab and M. T. zsu. Issues in data stream management. In *SIGMOD Rec.*, 32(2):5–14, 2003.

[4] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proceedings of the USENIX Annual Technical Conference 1998* pages 13–24.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *PODS 2002*, pages 1–16.

[6] C. E. Dyreson. Temporal coalescing with now granularity, and incomplete information. In *SIGMOD 2003*, pages 169–180.

[7] S. Babu and J. Widom. Continuous Queries Over Data Streams. *SIGMOD Record*, 30(3):109–120, Sept 2001.

[8] M. Gergatsoulis and Y. Stavrakas. Representing Changes in XML Documents Using Dimensions. In *Proceedings of XSym 2003*, pages 208–222.

[9] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. Query processing of streamed xml data. In *Proceedings of the eleventh International Conference on Information and Knowledge Management, CIKM 2002*, pages 126–133. November 2002.

[10] P. Buneman, S. Khanna, K. Tajima and W. C. Tan. Archiving Scientific Data. In *SIGMOD 2002*, pages 1–12.

[11] J. Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *the 9th International Workshop on Data Base Programming Languages (DBPL)*, Potsdam, Germany, September 2003.

[12] S. Bose, L. Fegaras, D. Levine, and V. Chaluvadi. A Query Algebra for Fragmented XML Stream Data. In *the 9th International Workshop on Data Base Programming Languages (DBPL)*, Potsdam, Germany, September 2003.

[13] P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes W3C Recommendation 02 May 2001, `http://www.w3.org/TR/xmlschema-2/`.

[14] D. Carney, U. Cetinternel, A. Rasin, S. B. Zdonik, M. Cherniack, M. Stonebraker. Operator Scheduling in a Data Stream Manager. In *VLDB 2003*, pages 838–849, 2003.

[15] R. Motwani, et al. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *the 1st Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, January 2003.

[16] S. Chandrasekaran, et al. TelegraphCQ: Continuous Data flow Processing for an Uncertain World. In *Proceedings of Conference on Innovative Data Syst. Res*, pages 269–280, 2003.

[17] A. Lerner and D. Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *VLDB 2003*, pages 345–356.

[18] D. Carney, et al. Monitoring streams—A New Class of Data Management Applications. In *VLDB 2002*, pages 215–226.

[19] B. Ludäscher, Y. Papakonstantinou, and P. Velikhov. Navigation-driven evaluation of virtual mediated views. In EDBT 2000, 7th International Conference on Extending Database Technology, Konstanz, Germany, March 27-31, 2000, pages 150–165.

[20] B. Babcock, S. Babu, R. Motwani and M. Datar. Monitoring streams—A New Class of Data Management Applications. In *SIGMOD 2003*, pages 253–264.

[21] F. Grandi and F. Mandreoli. The Valid Web: An XML/XSL Infrastructure for Temporal Management of Web Documents. In *Proceedings of International Conference on Advances in Information Systems*, pages 294–303, Izmir, Turkey, October 2000.

[22] Qizx/Open. http://www.xfra.net/qizxopen.

[23] A. Schmidt, F. Vaas, M. L. Kersten, M. J. Carey, I. Manolescu and R. Busse. XMark: A Benchmark for XML Data Management. In *VLDB 2002*, pages 974–985, 2002.