A Query Algebra for Fragmented XML Stream Data

Sujoe Bose   Leonidas Fegaras   David Levine   Vamsi Chaluvadi

Department of Computer Science and Engineering
The University of Texas at Arlington
416 Yates Street, P.O. Box 19015
Arlington, TX 76019-19015
{bose,fegaras,levine,chaluvad}@cse.uta.edu

Abstract

The increased usage of mobile devices coupled with an unprecedented demand for information has pushed the scalability problem of pull-based data service to the focus. A broadcast model of streaming data over a wireless medium has been proposed to be a viable alternative for information dissemination. In the streaming broadcast model, servers broadcast data in an asynchronous and unacknowledged mode while clients process personalized and complex queries locally, relieving the load on the server. We address the query processing of streamed XML data, which is fragmented into manageable chunks for easier synchronization. Although there has been some work done in defining algebras that model XQueries on XML documents, no work has been done in defining query algebras for fragmented XML stream data. We define a model for processing fragmented XML stream data, using the concept of *holes* and *fillers*. This model offers the flexibility required by the server to disseminate data in manageable fragments, whenever they become available, and to send repetitions, replacements and removal of fragments. We then present a query algebra for XQuery that operates on this streamed XML data model. The XML fragments are operated upon in a continuous pipelined fashion without the need of materializing the transmitted document at the client site.

Proceedings of the 9th International Conference on Data Base Programming Languages (DBPL), Potsdam, Germany, September 6-8, 2003

1 Introduction

1.1 Motivation

Traditionally, client-server architectures are service oriented, in which the servers process client requests in the form of queries over the server database and return the data pertaining to the client requests. In this setup, the onus is on the server to process the requests at the behest of the clients and to send back the results. The dramatic increase in the mobile devices and their data-providing applications, coupled with the complex and customized queries from the clients, may push the load on the servers to prohibitive proportions. The alternative method is to disseminate the data available at the servers (both static and real-time data) in a broadcast (or multicast) medium, with multiple channels of communication, on which the clients listen.

We envision a large class of applications for a push-based data model we call broadcast of streaming XML data, or *XStreamCast*. We assume that a typical configuration will consist of a small number of servers that transmit XML data over high-bandwidth streams and many clients that receive this 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. 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 enough memory capacity and processing power to evaluate sophisticated, continuous queries on the XML data stream. For example, a server may broadcast stock prices and a client may evaluate a continuous query on a wireless, mobile device that checks and warns (by activating a trigger) on rapid changes in selected stock prices within a time period.

We believe that there will be a growing interest in query processing of continuous data streams as the network bandwidth increases at rates higher than disk access times, which makes it inefficient to cache stream data on disk and process it using conventional database techniques. Moreover, 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, such as requesting a region from a geographical database. By distributing processing to clients, we reduce the server workload while increasing its availability. More importantly, when the push-based model is used, a client can correlate data between two or more sources by tuning-in and joining the data streams on-line, while, when the pull-based model is used, a client must submit one query to each server that holds the data and then join the results using a combining query. There are many sophisticated distributed query processing techniques that address the latter, such as semijoins, but, in general, large amounts of data must be transmitted from servers to clients before they are combined.

The standardization of XML, along with its widespread adoption and usage, presents it as the language of choice for communications between cooperating systems. XML, along with a rich set of supporting specifications and standards, has been the focus of current research. Even though there are several proposals for XML query languages [13], we adopt the XQuery standard proposed by World-Wide Web Consortium (W3C) [6]. While the advantages of continuous processing of streamed data compared to computation on stored data have been understood and realized [2, 3], not much work has been done in the streamed XML front with XQueries as the stream processing function. Much of the work related to XML stream processing has been focused on XML filtering, i.e., matching a large number of client XPath queries with XML documents at the server side and multicasting the matched documents to the appropriate clients [15, 1]. Although there have been some efforts on defining algebras that model XQueries on XML documents, to our knowledge, no work has been done in defining query algebras for streamed XML data.

As an example consider two XML streams: the *commodities* stream that lists the commodities by vendor along with its prices in USD:

```
<commodities>
   <vendor>
      <name> Wal-Mart </name>
        <items>
           <item>
              <name> PDA </name>
              <make> HP </make>
              <model> PalmPilot </model>
              <price currency="USD">315.25<price>
           </item> ...  </items>
   </vendor>
   ...
</commodities>
```

and the *currency* stream that provides currency conversion information:

```
<currencies>
   <currency>
      <code> USD </code>
         <name> US Dollars </name>
         <rates>
            <currency>
               <code> GBP </code>
               <buying> 1.34 </buying>
               <selling> 1.32 </selling>
            </currency> ... </rates>
   </currency>
   ...
</currencies>
```

We would like to issue a query to get a list of prices for the PalmPilots in GBP sorted by price, lowest to highest. The XQuery would look like:

```
<result>
{ for $c in stream(commodities)//vendor/items/item
  where $c/model/text() = "Palm-Pilot"
  return
    <vendor> $c/../../name/text() </vendor>
    <price>
      for $u in stream(currency)//rates/currency
      where $u/code/text() = "GBP"
      and $u/../../code/text()=$c/price/@currency
      return $u/buying/value()*$c/price/value()
    </price>
  sortby($c/price)
} </result>
```

The challenge is in evaluating this query against the XML data from the two streams. The implicit join between the commodities and the currency streams in the above query requires a state of unbounded length, since the GBP currency information may come at the end of the currency stream. Moreover, the sortby clause makes the query a blocking query, since no result can be obtained without reading both streams.

1.2 Our Approach

Since an XML stream can be visualized as an infinite sequence of data items belonging to some data set, in our framework, we fragment the stream into manageable chunks of information. These chunks, or fragments, are related to each other and can be reassembled at the client side upon arrival. However, the focus of our work is not in reconstructing the stream in the client devices, but rather to evaluate ad-hoc queries against these fragments at the client side in real time and produce results. Our XML fragments follow the *Hole-Filler model* 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. Note that a document can be fragmented to produce fillers and holes in arbitrary ways. Furthermore, the fragments can be arranged to any depth and breadth in a document tree.

Fragmenting a stream has the following advantages: While it is hard to synchronize on infinitely long streamed XML data, it is relatively easy to synchronize on fragmented XML data. The fragmentation also is necessary when pieces of data are made available to the server in real-time, so that information that is available may be transmitted as and when is received. For example, in a stock quote stream, the enumerated set of ticker symbols are near-static, however the stock quotes on those symbols change, and these changes need to be transmitted often as fragments, wrapped with sufficient context information. The stream fragments can be repeated for redundancy and to accommodate client devices that connect sporadically.

The novelty of this paper is in characterizing XML streams by a list of fragments, suitable for dissemination by servers and for processing by clients. This is captured in our streamed XML model by discrete fragments of XML data interrelated and intermittent in arrival, wherein a particular ordering of fragments is not guaranteed. This characterization allows servers to be able to repeat or replace fragments, introduce new ones or delete outdated ones. More importantly, this approach closely reflects the dynamics of data, especially in real-time applications, and provides the required controls on the server to make sure clients are synchronized with the data at the server. Another contribution of this paper is a novel query algebra based on the streamed XML model. This algebra commensurates with the main theme of our work: to directly evaluate the queries input by the client on streams of XML data without materializing it in memory. We extend our earlier work on an algebra for stored XML data [11] to operate on our streamed XML model rather than on stored XML documents. The challenge is in evaluating the operators against our streamed XML model, which consists of discrete hole and filler fragments interrelated to each other. Since, fragments are not guaranteed to arrive at a particular order, we need to take care of arrivals of filler fragments before the presence of a hole for the filler and vice versa. This leads to an algebra in which the evaluation on certain fragments may be suspended if the fragments have holes for which the fillers have not arrived and the path evaluation used in an algebraic operator leads to the hole.

1.3 Assumptions

In our push-based broadcasting framework, we make the following assumptions about the environment and the characteristics of the components involved. We assume that client devices connect into multiple data streams asynchronously and have reasonable amount of resources in terms of computing and storage. The communication medium is assumed to be noisy and lossy, but we will not address losses in the communication medium and handling a lossy medium is considered as future work. However, the fact that the stream fragments can be repeated, replaced, removed or added by the server, offsets the problem of the communication losses. It also enables clients to connect into the stream and be able to process the data stream. The server may send the fragments in any order. Another assumption is that data is well-typed. The server will transmit meta-data periodically to provide the structure of the XML data being transmitted (described in Section 2.1). The queries are assumed to be input through a menu interface or manually. The queries can be generic at first and can be refined to return desired results. Since clients can tune-in to streams at different points in time, and the servers disseminate fragments of data asynchronously, the results of queries will reflect the time in which the client are connected to the streams as well as the duration of the connections.

1.4 Layout of the Paper

In Section 2, we present a formal model for the streamed XML data, based on the Hole-Filler model, which serves as the basis for our query algebra. In Section 3, we review our earlier work on an algebra for stored XML data and, in Section 4, we extend this algebra to process streams of fragmented XML data. Our streamed query algebra operates on the fragments to produce continuous results in a pipelined fashion. Finally, we present the related work in Section 5 and we conclude in Section 6.

2 Our Model for Streamed XML Data

2.1 The Fragmented Hole-Filler Model

In our model, XML documents are transmitted in fragments. These fragments may be disseminated by a server in any order. To be able to relate fragments with each other, we introduce the concept of *holes* and *fillers*. A hole represents an empty node into which another rooted subtree, called a filler, could be positioned to complete the tree. The filler can in turn have holes in it, which will be filled by other fillers. Unique IDs are assigned to the holes. The fragment that corresponds to a hole has the same ID in its header. Thus, by substituting holes with the corresponding fillers, we can reconstruct the whole XML tree at the client side as it was in the server side. However, reconstructing the original XML tree is not always a good idea, since clients would have to wait for the end of the stream to begin processing. As will be discussed in the next section, our goal is to apply our query algebra on this streamed model and to process XML fragments as they become available.

At the server side, the XML document may be fragmented by recursively pruning the tree, inserting a hole at every point a tree is pruned and associating it with an ID. The server may prune the tree in an arbitrary way. This arrangement, of associating holes with fillers, takes care of out-of-sequence transmission, repetitions, replacements and removals. Another important piece of information transmitted by the server, is the *Tag Structure* of the document transmitted in the stream as a separate fragment. This tag structure is a structural summary that provides the structural make-up of the XML data and captures all the valid paths in the data. This information is useful while expanding wildcard path selections in the client queries. Moreover, this structure gives us the convenience of abbreviating the tag names with IDs (not used here) for compressing stream data. The following is a tag structure corresponding to the commodities stream:

```
<stream:structure>
   <tag name="commmodities" id="1">
       <tag name="vendor" id="2">
           <tag name="name" id="3"/>
           <tag name="items" id="4">
               <tag name="item" id="5">
                   <tag name="name" id="6"/>
                   <tag name="make" id="7"/>
                   <tag name="model" id="8"/>
                   <tag name="price" id="9"/>
               </tag>
           </tag>
       </tag>
   </tag>
</stream:structure>
```

As an example of the hole-filler model of fragmenting XML data, consider the commodities XML document stream with thousands of vendors, where each vendor potentially provides thousands of items. This document, in its entirety, may be too large to send as a single document. Moreover, data corresponding to all the vendors may not be available at the same time. The commodities stream would be transmitted in the following fragments, where each fragment is a filler for a hole in some other filler fragment:

Fragment 1:

```
<commodities>
   <vendor>
      <name> Wal-Mart </name>
        <items>
          <stream:hole id="10" tsid="5"/>
          <stream:hole id="20" tsid="5"/>
          ...
   </vendor> ... </commodities>
```

Fragment 2:

```
<stream:filler id="10" tsid="5">
  <item>
    <name> PDA </name>
    <make> HP </make>
    <model> PalmPilot </model>
    <price currency="USD">315.25<price>
  </item></stream:filler>
```

Fragment 3:

```
<stream:filler id="20" tsid="5">
  <item>
    <name> Calculator </name>
    <make> Casio </make>
    <model> FX-100 </model>
    <price currency="USD">50.25<price>
  </item></stream:filler>
```

Fillers are associated to holes by matching filler IDs with hole IDs. The filler with $id = 0$ is always the root filler, and hence the root of the fragmented document. The *tsid* attribute identifies the ID of the tag structure element corresponding to the filler fragment element. Using the tag structure ID, the structural context of the fragment can be deduced by looking-up the tag structure sent by the server intermittently. The filler of fragment 2 fills the hole with $id = 10$ inside fragment 1. It can be seen that fragment 1 is near-static while fragments 2 and 3 are not, due to variations in price. While fragment 1 can be transmitted earlier, the fragments 2 and 3 can be transmitted as soon as the price on an item is received. Note that the fragmentation can be done at any level in the XML tree, based on the flexibility in the granularity of fragmentation in various data domains. In the example above, we could have chosen to further fragment the right filler such that the "price" element, which is dynamic, be transmitted as a separate filler fragment.

2.2 Formal Definition of the Model

In this section, we formalize the fragmented stream model of XML data outlined in Section 2.1. The basic stream components transmitted by the server are fillers and holes, each with its own ID. In order for the server to be able to repeat, replace and remove fragments, the basic stream constructs are extended with repeat, replace and remove primitives. The server can also send an end-of-stream fragment, which is a marker for end of transmission. The stream content (γ) can be described by the following grammar:

γ ::= <filler id="m" tsid="n">x </filler>
| <repeat id="m" tsid="n">x </repeat>
| <replace id="m" tsid="n">x </replace>
| <remove id="m" tsid="n"/>
| <eos/>
| γ ; γ

where m and n are numbers, representing the *id* and *tsid* attributes respectively, and x is a valid XML fragment, defined as follows (XML attributes are ignored):

x ::= <tag>x </tag>
| <tag/>
| <hole id="m" tsid="n"/>
| PCDATA
| x x

(The root of the document is always transmitted with filler id = "0".) For convenience, we represent the stream components with the following syntactic shorthands, where m and n stand for the id and tsid attribute values respectively in the original form:

$$
\begin{array}{rcll}
\gamma & ::= & F(m,n,x) & \text{(a \textit{filler})} \\
& | & R(m,n,x) & \text{(a \textit{repeat})} \\
& | & U(m,n,x) & \text{(a \textit{replace})} \\
& | & D(m,n) & \text{(a \textit{remove})} \\
& | & eos & \text{(an \textit{end of stream})} \\
& | & \gamma \; ; \; \gamma & \text{(a \textit{stream sequence})}
\end{array}
$$

The stream components thus contain XML fragments of the original document and these XML fragments in turn may contain holes, which will be filled by other stream components. Note that we do not model silence during transmission, which is inherent in stream-based models. The reason is that we do not want to clutter the grammar and algebra with entities not contributing to the processing. We do however consider this aspect when evaluating certain operators that read from multiple streams, such as joins, since the streams may not be synchronized, or worse, one of the streams may be blocked.

2.3 Reconstruction of the XML Document from Stream Fragments

The fragmented XML document in the form of holes and fillers can be reconstructed, in its entirety, at the client by recursively filling the holes with the corresponding fillers. We define a stream transformation function $\mathcal{T}$, which maps XML fragments represented by γ onto an XML document. The stream transformation function involves the process of maintaining a mapping between filler IDs and the fillers, based on the stream components, and recursively filling holes with fillers with corresponding IDs. $\mathcal{T}$ can then be defined in terms of the filler mapping function $\mathcal{T}_1$, and the hole substitution function $\mathcal{T}_2$. To accurately depict the causal semantics of the stream primitives, which modify the mapping between filler IDs and the fillers, we represent function $\mathcal{T}_1$ as taking the stream as its argument, and returning an environment containing the mapping between the filler IDs and fillers. The hole substitution function $\mathcal{T}_2$ then substitutes the holes in the fragments recursively. Note that this function takes care of out-of-order arrival of holes and their corresponding fillers. After the holes are substituted, we have a set of XML fragments each representing various possible sub-trees, devoid of holes, of the original XML document. To retrieve the original document from this set, we retrieve the XML fragment with $id = 0$.

$\mathcal{T}$ is defined on a stream γ as $\mathcal{T}(\gamma) = \mathcal{T}_2(\mathcal{T}_1(\gamma, \emptyset))[0]$, where $\mathcal{T}_1$ returns an environment ζ, which is a set of bindings that, for each fragment, it binds the fragment ID to the fragment content. The filler mapping function, $\mathcal{T}$, is defined over the stream γ, and is shown in a case method of function definitions, over the possible stream primitives as follows:

$$
\begin{array}{rcl}
\mathcal{T}_1(F(m,n,x),\zeta) & = & \zeta \cup \{(m,x)\} \\
\mathcal{T}_1(R(m,n,x),\zeta) & = & \zeta \cup \{(m,x)\} \\
\mathcal{T}_1(D(m,n),\zeta) & = & \{\,(k,y) \,|\, (k,y) \in \zeta,\, k \neq m\,\} \\
\mathcal{T}_1(U(m,n,x),\zeta) & = & \mathcal{T}_1(D(m,n),\zeta) \cup \{(m,x)\} \\
\mathcal{T}_1(eos,\zeta) & = & \zeta \\
\mathcal{T}_1(\gamma_1 \,;\, \gamma_2,\zeta) & = & \mathcal{T}_1(\gamma_2, \mathcal{T}_1(\gamma_1,\zeta))
\end{array}
$$

Filler fragments and repeat fragments are appended into the environment ζ, while a delete fragment removes the mapping of an ID. A replace fragment causes the deletion of an existing filler with matching ID and the subsequent addition to the environment. A stream sequence is handled recursively and the end-of-stream fragment simply returns the final environment. The hole substitution function, $\mathcal{T}_2(\zeta)$ merges the fragments in ζ by filling holes with fillers:

$$
\begin{array}{rcl}
\mathcal{T}_2(\zeta) & = & \{\,(k, y[m/x]) \,|\, (m,x) \in \zeta,\, (k,y) \in \zeta, \\
& & \qquad \text{<hole id="m".../>} \in y\,\}
\end{array}
$$

where $y[m/x]$ replaces <hole id="m".../> in y with x. Finally, after applying $\mathcal{T}_2$ to ζ, we retrieve the entire XML tree by indexing the root (of id="0").

Thus, given a query $q(x_1, \ldots, x_n)$ over n XML trees $x_1, \ldots, x_n$, and a query $q'(\gamma_1, \ldots, \gamma_n)$ over n streams of XML data $\gamma_1, \ldots, \gamma_n$, we say that q and q' are equivalent iff:

$$
\forall \gamma_i : \quad \mathcal{T}(q'(\gamma_1, \ldots, \gamma_n)) \quad = \quad q(\mathcal{T}(\gamma_1), \ldots, \mathcal{T}(\gamma_n))
$$

that is, q and q' must return the same result for any input. We will use this natural transformation for proving the equivalence of our streamed XML algebra to our algebra for stored XML data. Our effort concentrates on designing q', operating on the fragments, such that it will yield identical result as q operating on entire XML documents. We present this in a two step process: first, we give the semantics of our algebra q, for stored XML data (XML trees). We then transform this algebra into q', which operates on the fragmented XML model and produces identical results as q. Reconstructing the original XML tree is not always a good idea, since clients would have to wait for the end of the stream to begin processing. The emphasis of our work is in providing the semantics of a query algebra such that, executing the query on the fragments and then constructing the resulting document to form the final output produces the same result as constructing the entire document and then executing the query. Thus the results can be pipelined and produced as and when they occur.

3 An Algebra for Stored XML data

In this section, we present our algebra for stored XML data introduced in earlier work [11]. Our streamed

$$
\begin{array}{rcl}
[\![\rho_v(T)]\!]_\delta & = & \{ < v = T > \} \\
[\![\sigma_{pred}(X)]\!]_\delta & = & \{ \, t \mid t \in [\![X]\!]_\delta ,\, [\![pred]\!]_{\delta \circ t} \, \} \\
[\![\pi_{v_1,\ldots,v_n}(X)]\!]_\delta & = & \{ < v_1 = t.v_1, \ldots, v_n = t.v_n > \; \mid t \in [\![X]\!]_\delta \, \} \\
[\![X \bigcup Y]\!]_\delta & = & [\![X]\!]_\delta +\!\!+ [\![Y]\!]_\delta \\
[\![X \bowtie_{pred} Y]\!]_\delta & = & \{ \, t_x \circ t_y \mid t_x \in [\![X]\!]_\delta ,\, t_y \in [\![Y]\!]_\delta ,\, [\![pred]\!]_{\delta \circ t_x \circ t_y} \, \} \\
[\![\mu^{v,path}_{pred}(X)]\!]_\delta & = & \{ \, t \circ < v = w > \mid t \in [\![X]\!]_\delta ,\, w \in \mathcal{P}(\delta \circ t, path),\, [\![pred]\!]_{\delta \circ t \circ < v = w >} \, \} \\
[\![\Delta^{\oplus,head}_{pred}(X)]\!]_\delta & = & \oplus / \{ \, [\![head]\!]_{\delta \circ t} \mid t \in [\![X]\!]_\delta ,\, [\![pred]\!]_{\delta \circ t} \, \} \\
[\![\Gamma^{v,\oplus,head}_{group,pred}(X)]\!]_\delta & = & \left\{ \begin{array}{l} \{ \, [\![group]\!]_{\delta \circ t_1} \circ < v = \oplus / \{ \, [\![head]\!]_{\delta \circ t_2} \mid t_2 \in [\![X]\!]_\delta ,\, [\![pred]\!]_{\delta \circ t_2} , \\ \qquad\qquad\qquad\qquad\qquad [\![group]\!]_{\delta \circ t_2} = [\![group]\!]_{\delta \circ t_1} \, \} > \\ \mid t_1 \in [\![X]\!]_\delta \, \} \end{array} \right.
\end{array}
$$

Figure 1: An Algebra for Stored XML Data

XML algebra, described in the next section, has the same algebraic operators but different data domains (streams rather than trees) and different semantics.

The algebraic bulk operators along with their semantics are given in Figure 1. The inputs and output of each operator are sequences of tuples that contain XML subtrees. These sequences are captured as lists of records and can be concatenated with list append, $+\!\!+$. There are other non-bulk operators, such as boolean comparisons, which are not listed here. The semantics is given in terms of record concatenation, $\circ$, and list comprehensions, $\{ \, e \mid \ldots \, \}$, which, unlike the set former notation, preserve the order and multiplicity of elements. The form $\oplus/\{ \cdots \}$ reduces the elements resulted from the list comprehension using the associative binary operator, $\oplus$ (a monoid, such as $\cup$, $+$, $*$, $\wedge$, $\vee$, etc). That is, for a non-bulk monoid $\oplus$, such as $+$, we have $+/\{a_1, a_2 \ldots, a_n\} = a_1 + a_2 + \cdots + a_n$, while for a bulk monoid, such as $\cup$, we have $\cup/\{a_1, a_2 \ldots, a_n\} = \{a_1\} \cup \{a_2\} \cup \cdots \cup \{a_n\}$.

The environment, δ, is the current record under consideration, and is used in the nested queries. Nested queries are mapped into an algebraic form in which some algebraic operators have predicates, headers, etc, that contain other algebraic operators. More specifically, for each record, δ, of the stream passing through the outer operator of a nested query, the inner query is evaluated by concatenating δ with each record of the inner query stream.

An unnest path is, and operator predicates may contain, a path expression $v/path$, $v/path/text()$, or $v/path/data()$, where v is a record attribute in the operator's input sequence and $path$ is a simple XPath of the form:

$$
\begin{array}{rcl}
path & ::= & A \\
 & \mid & A/path
\end{array}
$$

for a tag name A. The unnest operation is the only mechanism provided for traversing an XML tree structure. It uses function $\mathcal{P}$, which is defined over paths as follows:

$$
\begin{array}{rcl}
\mathcal{P}(t, v/path) & = & \mathcal{P}'(t.v, path) \\
\mathcal{P}'(\text{<A>}x \text{ </A>}, A/path) & = & \mathcal{P}'(x, path) \\
\mathcal{P}'(\text{<A>}x \text{ </A>}, A) & = & \{ \, \text{<A>}x \text{ </A>} \, \} \\
\mathcal{P}'(x_1 \, x_2, path) & = & \mathcal{P}'(x_1, path) \\
 & & \cup \, \mathcal{P}'(x_2, path) \\
\mathcal{P}'(t, path) & = & \emptyset \qquad \text{otherwise}
\end{array}
$$

The extraction operator, ρ, gets an XML document, T, and returns a singleton list whose unique element contains the entire XML tree. Selection (σ), projection (π), merging ($\cup$), and join ($\bowtie$) are similar to their relational algebra counterparts, while unnest (μ) and nest (Γ) are based on the nested relational algebra. The reduce operator, Δ, is used in producing the final result of a query/subquery, such as in aggregations and existential/universal quantifications. For example, the XML universal quantification **every** `$v` **in** `$x/A` **satisfies** `$v/A/data()>5` can be captured by the Δ operator, with $\oplus = \wedge$ and $head = v/A/data() > 5$. Collection query results can be constructed by Δ by using a collection monoid, such as $\cup$. Like the XQuery predicates, the predicates used in our XML algebraic operators have implicit existential semantics related to the (potentially multiple) values returned by path expressions. For example, the predicate $v/A/data() > 5$ used in the previous example has the implicit semantics $\exists x \in v/A/data() : x > 5$, since the path $v/A/data()$ may return more than one value. Finally, even though selections and projections can be expressed in terms of Δ, for convenience, they are treated as separate operations. Please refer to [11], for a complete treatment of translating queries written in XQuery to our algebra, and for some query normalization and optimization rules.

3.1 Example

To illustrate the usage of the operators in our algebra, we consider the following XQuery, which returns the list of vendors selling HP PDAs:

```
   for $b in document(commodities)//vendor//item
 where $b/name = "PDA"
   and $b/make = "HP"
return <vendor> { $b/../../name } </vendor>
```

Its corresponding equivalent algebraic form is:

$$\Delta^{\cup,h}(\sigma_{p_2}(\sigma_{p_1}(\mu^{b,v/\text{items/item}}(\mu^{v,c/\text{commodities/vendor}}(\rho_c(T))))))$$

where

$$\begin{aligned} T &= \text{document(commodities)} \\ h &= \text{element}(\text{“vendor”}, v/\text{name}) \\ p_1 &= b/\text{name} = \text{“PDA”} \\ p_2 &= b/\text{make} = \text{“HP”} \end{aligned}$$

The extraction operator, ρ_c, extracts the complete XML tree from the commodities document and binds it to the variable c, while the unnest operator, $\mu^{v,c/\text{commodities/vendor}}$, traverses the document using the path c/commodities/vendor and binds the elements obtained by the unnesting operation to the variable c. These elements are subsequently unnested by $\mu^{b,v/\text{items/item}}$, binding the variable b to each item. The selection operators, σ_{p_2} and σ_{p_1}, filter the elements based on the predicates b/name = "PDA" and b/make = "HP" respectively. The element construction function, element("vendor", v/name), constructs an XML element with tag name "vendor" and content v/name, while the reduce operator $\Delta^{\cup,h}$ unions together the elements returned by the element function.

4 An Algebra for Fragmented XML Stream Data

The algebraic operators for streamed XML data take stream fragments as input and produce stream fragments as output. The fragments are streamed through the various operators in a pipelined fashion. While all the incoming fragments are examined, some of them are discarded when the operator predicate evaluates to false. Some fragments may not contain enough information to be evaluated by a particular operator, due to the presence of holes. When an operator has insufficient information to validate a fragment, it suspends the processing of this fragment until the relevant fillers arrive.

Before we present the details of the algebraic operators, we give some useful definitions.

Since a streamed query may operate on multiple input streams, the client must maintain one tag structure for each input stream. For an input stream, s, function TS(s) returns the tag structure of s (an XML tree). Furthermore, for each input stream s and for each tsid m in $TS(s)$, function TSID(s, m) returns the subelement of this tag structure with id = m. Expressed in XPath, it is equal to:

$$\text{TSID}(s, m) = \text{TS}(s)//\text{tag}[@\text{id} = \text{“}m\text{”}]$$

For example, TSID(s, 7) evaluates to the tag structure element `<tag name="make" id="7"/>`, for the tag structure corresponding to the commodities stream s presented earlier. Given a tag structure s and a path expression $path$, $\mathcal{Q}(s, path)$ returns the set of tag structures reachable by $path$:

$$\begin{aligned} &\mathcal{Q}(\text{<tag name=“A” ...>}y\text{</tag>}, A/path) \\ &\qquad = \mathcal{Q}(y, path) \\ &\mathcal{Q}(\text{<tag name=“A” ...>}y\text{</tag>}, A) \\ &\qquad = \{\,\text{<tag name=“A” ...>}y\text{</tag>}\,\} \\ &\mathcal{Q}(x_1\, x_2, path) = \mathcal{Q}(x_1, path) \cup \mathcal{Q}(x_2, path) \\ &\mathcal{Q}(x, path) = \emptyset \qquad \text{otherwise} \end{aligned}$$

The intermediate results between our stream algebraic operators are records of the form: $<v_1 = e_1, \ldots, v_n = e_n>$, where v_i are range variables introduced by the algebraic operators and e_i are XML fragments. The hole ID m in $F(m, n, x)$ can be -1 to indicate that this filler is generated by one of the operators and does not fill any hole. Each range variable v is associated with a set of tag structures, called DOMAIN(v), which can be statically determined (at compile-time) from the algebraic operator tree and the tag structures of the input streams using the function $\mathcal{Q}$: Each range variable v ranges over the result of a path expression $w/path$, thus,

$$\text{DOMAIN}(v) = \{\, y \mid x \in \text{DOMAIN}(w),\ y \in \mathcal{Q}(x, path)\,\}$$

Content Restriction: For each intermediate result record of the form $< \ldots, v_i = F(m, n, x), \ldots >$, the condition, $\exists x \in \text{DOMAIN}(v_i) :\ x//\text{tag}/@\text{id} = \text{“}n\text{”}$, must hold. That is, the tag structure of the fragment streamed through the input and associated with the range variable v_i is not necessarily equal to one of the tag structures of v_i, but, more generally, it can be a descendant of one of them. If a record does not satisfy this restriction, it is suspended (explained below).

Each n-ary algebraic operator op in the algebraic tree of a query is associated with $n+1$ binding lists ζ_i, $0 \le i \le n$, one for for each input and one for the output, ζ_0, which maps fragment IDs to fragment contents. If a fragment from the operator's i'th input stream cannot be processed due to fragment holes, then the fragment is suspended in ζ_i. When a filler comes that fills one of the holes of a suspended fragment, then the fragment is awaken again and re-examined.

Function $\mathcal{P}'$ (defined in Section 3) is extended to handle holes:

$$\begin{aligned} \mathcal{P}'(\text{<hole id=“m” .../>}, path) &= \mathcal{P}'(\zeta_i(m), path) \\ &\qquad \text{if}\, m \in \zeta_i \\ \mathcal{P}'(\text{<hole .../>}, path) &= \{\,\perp\,\}\ \text{otherwise} \end{aligned}$$

That is, $\mathcal{P}'(x, path)$ returns a set of elements, where each element can be either a $\perp$ or an XML element. If there is at least one $\perp$ in $\mathcal{P}'(x, path)$, the *path* cannot be completely evaluated against x due to holes in x.

Given an intermediate result record t and a predicate *pred* over this record, $\mathcal{P}(t, pred)$ can return a true, false, or $\perp$ (undefined) value. That is, $\mathcal{P}'(x, path)$ returns a set of elements, where each element can be either a $\perp$ or an XML element. A predicate is in general a boolean formula that combines simple predicates using boolean operators ($\wedge$, $\vee$, etc). A simple predicate typically compares two XPaths or one XPath with a value. If *pred* is a simple predicate and there is a path $v/path$ in *pred* such that $\perp \in \mathcal{P}(t, v/path)$, then $\mathcal{P}(t, pred)$ evaluates to $\perp$; otherwise, $\mathcal{P}(t, pred)$ is true if for each path $v/path$ in *pred*, there is an XML value $x \in \mathcal{P}(t, v/path)$ such that *pred* evaluates to true when $v/path$ is replaced by x; otherwise, it is false. For the boolean operators, we use three-value logic. For example, false $\vee$ false = false, true $\vee\, x = x \,\vee$ true = true, otherwise $x \vee y = \perp$.

The extension of the path evaluation function $\mathcal{P}$ to handle fillers and holes forms the basis for the extension of our algebraic operators to operate on the fragmented XML model. We will now see how each operator is extended to handle the holes and fillers in XML streams. Note that each operator is producing output as a sequence of records as illustrated by the following BNF,

$$s := <v_1 = e_1, \ldots, v_n = e_n> \,;\, s \mid eos$$

Where $<v_1 = e_1, \ldots, v_n = e_n>$ is a record output from the operators, as discussed earlier in this section. This accounts for the pipelined operation of the algebraic operators while interrogating the stream, modeled as a sequence of fragments.

We will now show how some of the operators for stored XML algebra have been extended for streamed XML.

4.1 The Extraction Operator ρ_v

The extraction operator extracts the fragments from the stream and identifies each of these fragments with the range variable v. Every input fragment satisfies the content restriction since the tag structure of v is the root tag structure. Since this operator does not use paths, no fragments need to be suspended. The semantics of this operator is straightforward:

$$\begin{array}{rcl} [\![\rho_v(t\,;\,\gamma)]\!]_\delta & \rightarrow & < v = t >; [\![\rho_v(\gamma)]\!]_\delta \\ [\![\rho_v(eos)]\!]_\delta & \rightarrow & eos \end{array}$$

where the environment δ is the current record under consideration, as in the stored XML algebra.

4.2 The Selection Operator σ_{pred}

The selection operator filters the fragments from the stream based on the predicate specified. Since selection is a unary operator, it uses one environment ζ_1 for the input stream, which is always empty, and one environment ζ_0 for the output stream. If the selection predicate *pred* cannot be evaluated completely against the current tuple t of the input stream, then the tuple is suspended in ζ_0:

$$\zeta_0 \mathrel{+}= \{t\} \quad \text{if } \mathcal{P}(\delta \circ t, pred) = \perp$$

where operation $\zeta_0 \mathrel{+}= ts$ adds a set of tuples ts to ζ_0. Thus, selection lets the tuple t pass to the output stream only if *pred* evaluates to true:

$$\begin{array}{rcl} [\![\sigma_{pred}(t\,;\,\gamma)]\!]_\delta & = & \text{if } \mathcal{P}(\delta \circ t, pred) = \text{true} \\ & & \quad \text{then } t\,;\, [\![\sigma_{pred}(\gamma)]\!]_\delta \\ & & \quad \text{else } [\![\sigma_{pred}(\gamma)]\!]_\delta \\ [\![\sigma_{pred}(eos)]\!]_\delta & = & eos \end{array}$$

If $\mathcal{P}(\delta \circ t, pred)$ evaluates to $true$, then the fragment record t is output. The operator then recursively processes the remaining stream of records. If $\mathcal{P}(\delta \circ t, pred)$ evaluates to false (failed predicate) or $\perp$ (presence of a hole), then the fragment record t is not embedded to the output stream. Additionally, at the arrival of eos, the buffer ζ_0 is cleared, i.e $\zeta_0 = \emptyset$.

4.3 The Projection Operator $\pi_{v_1,\ldots,v_n}$

The projection operator uses the projection variables to select attributes from the current tuple of the input stream:

$$\begin{array}{l} [\![\pi_{v_1,\ldots,v_n}(t\,;\,\gamma)]\!]_\delta \\ \quad = < v_1 = t.v_1, \ldots, v_n = t.v_n >; [\![\pi_{v_1,\ldots,v_n}(\gamma)]\!]_\delta \\ [\![\pi_{v_1,\ldots,v_n}(eos)]\!]_\delta = eos \end{array}$$

4.4 The Merge Operator $\bigcup$

The merge operator propagates the input tuples from either one of the two input streams into the output stream (without removing duplicates):

$$\begin{array}{rcl} [\![(t_1\,;\,\gamma_1) \bigcup \gamma_2]\!]_\delta & = & t_1\,;\, [\![\gamma_1 \bigcup \gamma_2]\!]_\delta \\ [\![\gamma_1 \bigcup (t_2\,;\,\gamma_2)]\!]_\delta & = & t_2\,;\, [\![\gamma_1 \bigcup \gamma_2]\!]_\delta \\ [\![eos \bigcup \gamma]\!]_\delta & = & [\![\gamma \bigcup eos]\!]_\delta \;=\; \gamma \end{array}$$

Note that the semantic interpretation of the merge operation captures effectively the processing of streamed fragments when data is available in either of the streams, or possibly on both at the same time.

4.5 The Join Operator $\bowtie_{pred}$

The streamed join operator performs a join between the fragments of two streams γ_1 and γ_2 based on the predicate *pred*. It suspends all the tuples from both input streams because each fragment from either stream needs to be preserved since it may be joined with incoming fragments from the other stream. If the evaluation of the join predicate *pred* over a pair of records

t_1 and t_2 is undetermined (i.e., $\perp$) due to the presence of holes, the concatenation $t_1 \circ t_2$ is suspended in the binding list of the output stream, ζ_0. Furthermore, tuples from the left stream are suspended in ζ_1:

$$\begin{aligned} &[\![(t_1\,;\,\gamma_1) \bowtie_{pred} \gamma_2]\!]_\delta \\ &\quad = \{\ t_1 \circ t_2 \mid t_2 \in \zeta_2,\ \mathcal{P}(\delta \circ t_1 \circ t_2, pred) = \text{true}\ \} \\ &\qquad\qquad ;\ [\![\gamma_1 \bowtie_{pred} \gamma_2]\!]_\delta \end{aligned}$$

$$\begin{aligned} &\zeta_1 \mathrel{+}= \ t_1 \\ &\zeta_0 \mathrel{+}= \ \{\ t_1 \circ t_2 \mid t_2 \in \zeta_2,\ \mathcal{P}(\delta \circ t_1 \circ t_2, pred) = \perp\ \} \end{aligned}$$

The tuples from the right stream are suspended in ζ_2:

$$\begin{aligned} &[\![\gamma_1 \bowtie_{pred} (t_2\,;\,\gamma_2)]\!]_\delta \\ &\quad = \{\ t_1 \circ t_2 \mid t_1 \in \zeta_1,\ \mathcal{P}(\delta \circ t_1 \circ t_2, pred) = \text{true}\ \} \\ &\qquad\qquad ;\ [\![\gamma_1 \bowtie_{pred} \gamma_2]\!]_\delta \end{aligned}$$

$$\begin{aligned} &\zeta_2 \mathrel{+}= \ t_2 \\ &\zeta_0 \mathrel{+}= \ \{\ t_1 \circ t_2 \mid t_1 \in \zeta_1,\ \mathcal{P}(\delta \circ t_1 \circ t_2, pred) = \perp\ \} \end{aligned}$$

When an *eos* is received in stream γ_1, the contents of buffer ζ_2 can be flushed, since they are not needed anymore (there cannot be any more fragments of XML from stream ζ_1 that would need to be joined with fragments in ζ_2). The converse is true when an *eos* is received in stream γ_2, wherein buffer ζ_1 can be flushed. After both streams receive an *eos*, an *eos* is embedded to the output stream. Finally, and more importantly, the binding lists ζ_0 is reduced when a filler of a hole in a fragment in ζ_0 arrives from the input streams, which causes the tuple that contains this fragment to be reconsidered by the join operator.

The semantics of our join operator resembles the symmetric join algorithm [23]. To evaluate the equijoin $X \bowtie_{x.A=y.B} Y$ using a symmetric join, we have to maintain two hash tables in memory that contain the same number of buckets, one for stream X with a hash function based on $x.A$ and one for the stream Y based on $y.B$. When a tuple, x, arrives in the stream X, it is inserted in the X hash table and is joined with all the tuples in the Y hash table that have the same hash key. The matched pairs are immediately streamed into the output. A similar operation is performed when a new tuple arrives in the stream Y.

4.6 The Unnest Operator $\mu^{v,path}_{pred}$

The unnest operator provides the means to traverse the tree based on the path expression *path* and the predicate *pred*. It unnests the XML fragments based on the path expression and then applies the predicate validation:

$$\begin{aligned} &[\![\mu^{v,path}_{pred}(t\,;\,\gamma)]\!]_\delta \\ &\quad = \{\ t \circ < v = w > \mid w \in \mathcal{P}(\delta \circ t, path),\ w \neq \perp, \\ &\qquad\qquad \mathcal{P}(\delta \circ t \circ < v = w >, pred) = \text{true}\ \} \\ &\qquad\quad ;\ [\![\mu^{v,path}_{pred}(\gamma)]\!]_\delta \end{aligned}$$

$$\begin{aligned} \zeta_0 \mathrel{+}= \ &\{\ t \circ < v = w > \mid w \in \mathcal{P}(\delta \circ t, path), \\ &\quad \mathcal{P}(\delta \circ t \circ < v = w >, pred) = \perp\ \} \end{aligned}$$

$$[\![\mu^{v,path}_{pred}(eos)]\!]_\delta = eos$$

If the unnesting path or the predicate evaluates to $\perp$, then the unnesting pair (the current record concatenated with one of the results of applying *path* over the record) is suspended in ζ_0. If an *eos* is received, we return it and we flush the output buffer ζ_0.

4.7 The Reduce Operator $\Delta^{\oplus,head}_{pred}$

The reduce operator applies the header to each tuple and merges the results using $\oplus$:

$$\begin{aligned} &[\![\Delta^{\oplus,head}_{pred}(t\,;\,\gamma)]\!]_\delta \\ &\quad = \text{if } \mathcal{P}(\delta \circ t, pred) = \text{true} \ \wedge\ \mathcal{P}(\delta \circ t, head) \neq \perp \\ &\qquad\quad \text{then } \mathcal{P}(\delta \circ t, head)\ \oplus\ [\![\Delta^{\oplus,head}_{pred}(\gamma)]\!]_\delta \\ &\qquad\quad \text{else } [\![\Delta^{\oplus,head}_{pred}(\gamma)]\!]_\delta \end{aligned}$$

$$[\![\Delta^{\oplus,head}_{pred}(t\,;\,\gamma)]\!]_\delta = eos$$

If either the predicate or the header is unable to evaluate, the current tuple is suspended:

$$\zeta_0 \mathrel{+}= \ \{t\} \quad \text{if } \mathcal{P}(\delta \circ t, pred) = \perp \vee \mathcal{P}(\delta \circ t, head) = \perp$$

4.8 The Nest Operator $\Gamma^{v,\oplus,head}_{group,pred}$

The nest operator itself does not produce any output, since it is blocked:

$$\begin{aligned} [\![\Gamma^{v,\oplus,head}_{group,pred}(t\,;\,\gamma)]\!]_\delta &= [\![\Gamma^{v,\oplus,head}_{group,pred}(\gamma)]\!]_\delta \\ [\![\Gamma^{v,\oplus,head}_{group,pred}(eos)]\!]_\delta &= eos \end{aligned}$$

If any of the head, group, or pred cannot be evaluated completely, the tuple t is suspended in ζ_1. The ζ_0 binding list contains one tuple for each group. If a new group is found, then a new tuple is inserted in ζ_0; otherwise, the head of the current tuple is accumulated in the group tuple:

$$\begin{aligned} &\text{if } \mathcal{P}(\delta \circ t, pred) = \text{true:} \\ &\quad \text{find } x \in \zeta_0 :\ \mathcal{P}(\delta \circ x, group) = \mathcal{P}(\delta \circ t, group); \\ &\quad \text{if none exists:} \\ &\qquad \text{then } \zeta_0 \mathrel{+}= \ \mathcal{P}(\delta \circ t, group) \circ < v = \mathcal{P}(\delta \circ t, head) > \\ &\qquad \text{else } x.v\ =\ x.v\ \oplus\ \mathcal{P}(\delta \circ t, head). \end{aligned}$$

The groups can only be flushed after the end of the input stream.

4.9 An XQuery Example Using the Streamed XML Algebra

To illustrate the usage of the operators in our streamed XML algebra, we consider the XQuery presented in Section 3.1, which returns the list of names of vendors selling HP PDAs evaluated against the fragments from the commodities stream, as shown in Section 2. Let f_1, f_2, f_3 correspond to the fragments 1, 2 and 3. The commodities stream can be visualized as a sequence of the fragments $f_1; f_2; f_3; ...; eos$. When the fragments are streamed through the operators, the unnest operator, $\mu^{v,\mathrm{c/commodities/vendor}}$, suspends the fragment f_1, since it encounters a hole with $id = 10$ during its path traversal. When the fragment f_2 arrives, the suspended fragment is processed, and is streamed through. This fragment is them passed through the second unnest operator as well as the selection operators, σ_{p_2} and σ_{p_1}, and is added to the result stream. When the fragment f_3 arrives, the fragment f_1, still being suspended due to another unresolved hole with $id = 20$, is similarly streamed through the unnest operator, It is however filtered out in the selection stage. The *eos* stream element flushes the buffers. Note that the structural context of a fragment can be deduced with the help of the *tsid* attribute in the fragments and the tag structures transmitted by the server.

4.10 Equivalence Between the Stored and the Streamed XML Algebra

Given an XML query, its algebraic tree based on our streamed XML algebra is exactly the same as that for our stored XML algebra, since both algebras use identical algebraic operators. Therefore, to prove the equivalence theorem in Section 2.3, we need simply to prove that for each n-ary algebraic operator *op* of the stored algebra and its equivalent *op*$'$ of the streamed algebra, we have:

$$op(\mathcal{T}(\gamma_1), \ldots, \mathcal{T}(\gamma_n)) \;=\; \mathcal{T}(op'(\gamma_1, \ldots, \gamma_n))$$

for arbitrary finite streams γ_i. That way, by starting from the algebraic tree in the stored algebra of an n-ary query q, we can build the algebraic tree in the streamed algebra using the above natural transformations, since the $\mathcal{T}$ translations can propagate bottom-up in the query tree, yielding at the end the algebraic tree for the streamed query q' that satisfies:

$$q(\mathcal{T}(\gamma_1), \ldots, \mathcal{T}(\gamma_n)) \;=\; \mathcal{T}(q'(\gamma_1, \ldots, \gamma_n))$$

We leave the proofs for an extended version of this paper. With the fragmented model, it is beneficial to operate on fragments as and when they arrive, instead of waiting to materialize the complete document for two main reasons, firstly, processing can be continuous, pipelined and timely, secondly, the fragments that do not satisfy the predicates can be safely discarded as soon as possible thereby conserving memory.

5 Related Work

There are many recent projects related to query processing on data streams, which are overviewed elsewhere [2, 3].

The hole-filler model for XML data has been proposed in [19] 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 model, for fragmenting XML data to be sent to clients for selective query processing. Our main motivation is to relieve the load on the server and leverage the processing power in client devices. Based on previous experience with traditional databases, queries can be optimized more effectively if they are first translated into a suitable internal form with clear semantics, such as an algebra or calculus. There are already many proposals for algebras on semi-structured and XML data, including an algebra based on structural recursion [5], YATL [9, 8], SAL [4], x-algebra [12], TAX [17], and the Niagara algebra [22]. In contrast to these algebras, with the possible exception of the Niagara algebra, our work is based on the nested relational algebra, since our focus is on pipelining the algebraic operators using main-memory, relational evaluation algorithms. In addition, there is a recent proposal for a data model and an XQuery algebra by the W3C committee [10], whose main purpose is in expressing well-formed semantics, such as type inference. This algebra is basically a core subset of XQuery and, unlike conventional database algebras, does not address performance issues.

Our goals are similar to those of the Niagara Continuous Query Processing project, NiagaraCQ [7]. A recent work by this group proposed the use of punctuations for handling blocking operators [20]. A punctuation is a hint transmitted by a server to clients along with the data to indicate properties about the data. One example of a punctuation is the indication that all prices of stocks starting with 'A' have already been transmitted. Punctuations are properties that hold from the point of their transmission up-to the end of the stream, allowing us to view an infinite stream as a mixture of finite streams.

There is some work recently on using stream transducers to process continuous XML streams [18]. A transducer generalizes deterministic finite automata (DFAs) in that it allows the generation of output during a state transition. Even though DFAs have been shown to achieve a high throughput for XPath expressions and for non-blocking XQueries on single XML streams [14], it still to be shown that are also effective for multiple input streams, which require advanced main-memory join techniques that have already been successfully addressed by main-memory databases.

6 Conclusion

In this paper, we have illustrated the main theme of our work: evaluating the user defined queries on fragmented XML stream data in a continuous fashion, as opposed to materializing the stream and then evaluating the queries on the complete XML. We started by motivating the need for fragmented XML dissemination and subsequent query evaluation on streamed fragments. Our query algebra for processing of fragmented XML data is modeled in the lines of that for complete XML data, hence the query optimizations proposed in our previous work can be applied to the fragmented XML data processing as well. The streamed fragments are processed by our algebraic operators and are then combined to form the final result. Since the data is continuous and so are the queries, we continuously update the result to reflect the data and the query predicates.

References

[1] M. Altinel and M. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *VLDB 2000*.

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

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

[4] C. Beeri and Y. Tzaban. SAL: An Algebra for Semistructured Data and XML. In *WebDB 1999*, pages 37–42.

[5] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *SIGMOD 1996*, pages 505–516.

[6] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery: A Query Language for XML. W3C Working Draft. Available at `http://www.w3.org/TR/xquery/`, 2000.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD 2000*, pages 379–390.

[8] V. Christophides, S. Cluet, and J. Siméon. On Wrapping Query Languages and Efficient XML Integration. In *SIGMOD 2000*, pages 141–152.

[9] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your Mediators Need Data Conversion! In *SIGMOD 1998*, pages 177–188.

[10] D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Simeon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft. Available at `http://www.w3.org/TR/query-algebra/`, 2002.

[11] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. Query Processing of Streamed XML Data. In *CIKM 2002*, pages 126–133.

[12] M. Fernandez, J. Simeon, and P. Wadler. An Algebra for XML Query. In *FST TCS 2000*.

[13] D. Florescu, A. Levy, and A. Mendelzon. Database Techniques for the World-Wide Web: A Survey. *SIGMOD Record*, 27(3):59–74, 1998.

[14] T. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. In *ICDE 2003*.

[15] A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *SIGMOD 2003*, pages 419–430.

[16] Z. Ives, A. Levy, and D. Weld. Efficient Evaluation of Regular Path Expressions on Streaming XML Data. Technical Report, University of Washington, 2000, UW-CSE-2000-05-02.

[17] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A Tree Algebra for XML. In *DBPL 2001*, pages 149–164.

[18] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *VLDB 2002*.

[19] B. Ludäscher, Y. Papakonstantinou, and P. Velikhov. Navigation-driven Evaluation of Virtual Mediated Views. In *EDBT 2000*, LNCS 1777.

[20] P. Tucker, D. Maier, T. Sheard, and L. Fegaras. Online analysis and Querying of Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, May-June 2003.

[21] T. Urhan and M. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 23(3):27–33, 2000.

[22] S. Viglas, L. Galanis, D. DeWitt, D. Maier, and J. Naughton. Putting XML Query Algebras into Context. Unpublished manuscript, 2002.

[23] A. Wilschut and P. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *PDIS 1991*, pages 68–77.