Query Processing of Streamed XML Data

Leonidas Fegaras, David Levine, Sujoe Bose, and 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
email: fegaras@cse.uta.edu

ABSTRACT

We are addressing the efficient processing of continuous XML streams, in which the server broadcasts XML data to multiple clients concurrently through a multicast data stream, while each client is fully responsible for processing the stream. In our framework, a server may disseminate XML fragments from multiple documents in the same stream, can repeat or replace fragments, and can introduce new fragments or delete invalid ones. A client uses a light-weight database based on our proposed XML algebra to cache stream data and to evaluate XML queries against these data. The synchronization between clients and servers is achieved through annotations and punctuations transmitted along with the data streams. We are presenting a framework for processing XML queries in XQuery form over continuous XML streams. Our framework is based on a novel XML algebra and a new algebraic optimization framework based on query decorrelation, which is essential for non-blocking stream processing.

Categories and Subject Descriptors

H.3.3 Information Search and Retrieval—Query formulation, H.2.4 Systems—Query processing [

General Terms

]: Languages, Performance

Keywords

XML, Databases, Query Optimization, Query Processing

1. INTRODUCTION

1.1 Motivation

XML [17] has emerged as the leading textual language for representing and exchanging data on the web. Even though HTML is still the dominant format for publishing documents on the web, XML has become the prevalent exchange format for business-to-business transactions and for enterprise Intranets. It is expected that in the near future the Internet will be populated with a vast number of web-accessible XML files. One of the reasons of its popularity is that, by supporting simple nested structures of tagged elements, the XML format is able to represent both the structure and the content of complex data very effectively. To take advantage of the structure of XML documents, new query languages had to be invented that go beyond the simple keyword-based boolean formulas supported by current web search engines.

There are several recent proposals for XML query languages [13], but none has been adopted as a standard yet. Nevertheless, there is a recent working draft of an XML query language released by the World-Wide Web Consortium (W3C), called XQuery [6], which may become a standard in the near future. The basic features of XQuery are illustrated by the following query (taken from a W3C web page [17]):

```
<bib>{
    for $b in document("http://www.bn.com")/bib/book
 where $b/publisher = "Addison-Wesley"
   and $b/@year >1991
return <book year={ $b/@year }> { $b//title } </book>
} </bib>
```

which lists books published by Addison-Wesley after 1991, including their year and title. The document(*URL*) expression returns an entry point to the XML data contained in the XML document located at the specified URL address. XQuery uses path expressions to navigate through XML data. For example, the tag selection, $b/publisher, returns all the children of $b with tag name, publisher, while the wildcard selection, $b//title, returns all the descendants of $b (possibly including $b itself) with tag name, title. Influenced by modern database query languages, such as the OQL language of the ODMG standard, XQuery allows complex queries to be composed from simpler ones and supports many advanced features for navigating and restructuring XML data, such as XML data construction, aggregations, universal and existential quantification, and sorting.

There have been many commercial products recently that take advantage of the already established database management technology for storing and retrieving XML data, although none of them fully supports XQuery yet. In fact, nearly all relational database vendors now provide some functionality for storing and handling XML data in their systems. Most of these systems support automatic inser-

CIKM'02, November 4–9, 2002, McLean, Virginia, USA.

tion of canonically-structured XML data into tables, rather than utilizing the XML schemas for generating application-specific database schemas. They also provide methods for exporting database data into XML form as well as querying and transforming these forms into HTML.

The effective processing of XML data requires some of the storage and access functionality provided by modern database systems, such as query processing and optimization, concurrency control, integrity, recovery, distribution, and security. Unlike conventional databases, XML data may be stored in various forms, such as in their native form (as text documents), in a semi-structured database conforming to a standard schema, or in an application-specific database that exports its data in XML form, and may be disseminated from servers to clients in various ways, such as by broadcasting data to multiple clients as XML streams or making them available upon request.

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. If XML data are stored in an application-specific database schema, then XML queries over these data can be translated into the native query language supported by the database system, which in turn can be compiled into its native algebraic form. A major research issue related to this approach, which has been addressed in our previous work [10], is the automatic generation of the database schema as well as the automatic translation of XML queries into database queries over the generated schema. On the other hand, if XML data are stored as semi-structured data or processed on the fly as XML streams, then a new algebra is needed that captures the heterogeneity and the irregularities intrinsic to XML data [4]. In fact, there are already algebras for semi-structured data, including an algebra based on structural recursion [4], YATL [8, 7], SAL [3], x-algebra [12], and x-scan [15]. None of these algebras have been used for the emerging XML query languages yet, and there is little work on algebraic query optimization based on these algebras.

This paper is focused on the efficient processing of continuous XML streams. Most current web servers adopt the "pull-based" processing technology in which a client submits a query to a server, the server evaluates the query against a local database, and sends the result back to the client. An alternative approach is the "push-based" processing, in which the server broadcasts parts or the entire local database to multiple clients through a multicast data stream usually with no acknowledge or handshaking, while each client is fully responsible for processing the stream [1]. 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 [2], 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. Stream data may be infinite and transmitted in a continuous stream, such as measurement or sensor data transmitted by a real-time monitoring system continuously.

We would like to support the combination of both processing technologies, pull and push, in the same framework. The unit of transmission in an XML stream is an XML fragment, which corresponds to one XML element from the transmitted document. In our framework, a server may prune fragments from the document XML tree, called *fillers*, and replace them by *holes*. A hole is simply a reference to a filler (through a unique ID). Fillers, in turn, may be pruned into more fillers, which are replaced by holes, and so on. The result is a sequence of small fragments that can be assembled at the client side by filling holes with fillers. The 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 client is responsible for caching and reconstructing parts of the original XML data in its limited memory (if necessary) and for evaluating XML queries against these data. Like relational database data, stream data can be processed by relational operators. Unlike stored database data though, which may provide fast access paths, the stream content can only be accessed sequentially. Nevertheless, we would like to utilize the same database technology used in pulling data from a server for pushing data to clients and for processing these data at the client side with a light-weight database management system. 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. Since stock prices may change through time, the client must be able to cache (in main memory or on secondary storage) old prices.

Since XML fragments are processed at the client side as they become available, the query processing system resembles a main-memory database, rather than a traditional, secondary-storage-based database. Although a main-memory database has a different back-end from a secondary-storage database, main-memory queries can be mapped to similar algebraic forms and optimized in a similar way as traditional queries. One difficult problem of processing algebraic operators against continuous streams is the presence of blocking operators, such as sorting and group-by, which require the processing of the entire stream before generating the first result. Processing these operators effectively requires that the server passes some hints, called *punctuations*, 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. This is very valuable information to a client that performs a group-by over the stock names because it can complete and flush from memory all the groups that correspond to these stock names. Extending a query processor to make an effective use of such punctuations has not yet been investigated by others.

1.2 Our Approach

This paper addresses the efficient processing of continuous XML streams, in which a server broadcasts XML data to multiple clients concurrently. In our framework, a server may disseminate XML fragments from multiple documents in the same stream, can repeat or replace fragments, and can introduce new fragments or delete invalid ones. In our approach, a client uses a light-weight, in-memory database to cache stream data and physical algorithms based on an

XML algebra to evaluate XML queries against these data. The synchronization between clients and servers is achieved through annotations and punctuations transmitted along with the data streams. A necessary information needed by a client is the structure of the transmitted XML document, called the *Tag Structure*. The server periodically disseminates this Tag Structure as a special annotation in XML form. For example, the following Tag Structure:

```
<stream:structure>
   <tag name="bib" id="1">
       <tag name="vendor" id="2" attributes="id">
           <tag name="name" id="3"/>
           <tag name="email" id="4"/>
           <tag name="book" id="5"
                     attributes="ISBN related_to">
               <tag name="title" id="6"/>
               <tag name="publisher" id="7"/>
               <tag name="year" id="8"/>
               <tag name="price" id="9"/>
               <tag name="author" id="10">
                   <tag name="firstname" id="11"/>
                   <tag name="lastname" id="12"/>
               </tag> </tag> </tag> </tag>
</stream:structure>
```

corresponds to the following partial DTD:

```
<!ELEMENT bib (vendor*)>
<!ELEMENT vendor (name, email, book*)>
<!ATTLIST vendor id ID #REQUIRED>
<!ELEMENT book (title, publisher?, year?, price, author+)>
<!ATTLIST book ISBN ID #REQUIRED>
<!ATTLIST book related_to IDrefs>
<!ELEMENT author (firstname?, lastname)>
```

The Tag Structure is derived by the server from the data, rather than the type, and summarizes all the valid paths to the data. For example, according to the above Tag Structure, `/bib/vendor/name` is a valid path. The Tag Structure is used, among other things, to resolve wildcard selections in client-side XQueries. It also allows the compression of the transmitted data and annotations by replacing tag names with id numbers. The Tag Structure is always finite even in the case of recursive data types, such as the part-subpart hierarchy, since even if the data stream is infinite, the nesting level of the actual data cannot grow infinitely. Tag Structure annotations are very important and must be repeated very often by the server to cope with those clients who connect to the stream asynchronously at undetermined times.

The most important annotations in our framework are fillers and holes. A filler has a unique hole id, hid, referenced by hole, and a Tag Structure id (tsid):

```
<stream:filler hid="1234" tsid="5">
     XML element
</stream:filler>
```

which introduces a new book with ID 1234. It is a book because the Tag Structure id is 5, which means that it can be reached from the path /bib/vendor/book. A hole can take one of the following forms:

```
<stream:hole hid="1234"/>
<stream:hole hid="100 199"/>
```

The first form corresponds to one filler, namely to that with hid 1234, while the second form corresponds to 100 fillers, those whose hid's range from 100 to 199.

In our framework, fragments may be repeated, replaced, or removed using the following annotations:

```
<stream:repeat hid="1234"> ... </stream:repeat>
<stream:replace hid="1234"> ... </stream:replace>
<stream:remove hid="1234"/>
```

Query processing is performed at the client side with the help of a light-weight query optimizer. A major component of a query optimizer is an effective algebra, which would serve as an intermediate form from the translation of abstract queries to concrete evaluation algorithms. We are presenting a new XML algebra and a query optimization framework based on query normalization and query unnesting (also known as query decorrelation). There are many proposals on query optimization that are focused on unnesting nested queries [**?**, 9]. Nested queries appear more often in XML queries than in relational queries, because most XML query languages, including XQuery, allow complex expressions at any point in a query. Current commercial database systems typically evaluate nested queries in a nested-loop fashion, which is unacceptable for on-line stream processing and does not leave many opportunities for optimization. Most proposed unnesting techniques require the use of outer-joins, to prevent loss of data, and grouping, to accumulate the data and to remove the null values introduced by the outer-joins. If considered in isolation, query unnesting itself does not result in performance improvement. Instead, it makes possible other optimizations, which otherwise would not be possible. More specifically, without unnesting, the only choice of evaluating nested queries is a naive nested-loop method: for each step of the outer query, all the steps of the inner query need to be executed. Query unnesting promotes all the operators of the inner query into the operators of the outer query. This operator mixing allows other optimization techniques to take place, such as the rearrangement of operators to minimize cost and the free movement of selection predicates between inner and outer operators, which enables operators to be more selective. Our XML query unnesting method is influenced by the query unnesting method for OODB queries presented in our earlier work [11].

If the data stream were finite and its size were smaller than the client buffer size, then the obvious way to answer XQueries against an XML stream is to reconstruct the entire XML document in memory and then evaluate the query against the cached document. But this is not a realistic assumption even for finite streams. Client computers, which are often mobile, have typically limited resources and computing power. After XQueries are translated to the XML algebra, each algebraic operator is assigned a stream-based evaluation algorithm, which does not require to consume the entire stream before it produces any output. Even when a data stream is finite, some operations, such as sorting and group-by with aggregation, may take too long to complete. However, clients may be simply satisfied with partial results, such as the average values of a small sample of the data rather than the entire database. Online aggregation [14] has addressed this problem by displaying the progress at any point of time along with the accuracy of the result and by allowing the client to interrupt the aggregation process. Our approach is based on annotations about the data con-

tent, called *punctuations*. A punctuation is a hint sent by the server to indicate a property about the data already transmitted. This hint takes the form of a predicate that compares a path expression, which uses tsid's rather than tag names, with a constant value, as follows:

```
<stream:punctuation property="path cmp constant"/>
```

For example, the following punctuation

```
<stream:punctuation
   property="/bib/vendor[3]/book/@ISBN <= 1000"/>
```

indicates that the server has already transmitted all books published by the 3rd vendor whose ISBN is less than or equal to 1000. We are presenting a set of stream-based evaluation algorithms for our XML algebraic operators that make use of these punctuations to reduce the amount of resources needed by clients to process streamed data. For example, suppose that a client joins the above stream of data about books with the stream of book prices provided by Amazon.com using the ISBN as a cross-reference. Then one way to evaluate this join is using an in-memory hash join [16] where both the build and probe tables reside in memory. When ISBN punctuations are received from both streams, some of the hash buckets in both hash tables can be flushed from memory since, according to the punctuations received, they will not be used again. In our framework, all punctuations are stored in a central repository at the client side and are being consulted on demand, that is, only when the local buffer space of an evaluation operator overflows.

The rest of the paper is organized as follows. Section 2 presents a new XML algebra and a new algebraic optimization framework based on query decorrelation. It also presents translation rules for compiling XQuery into an algebraic form. Our XML algebra can be used generically for processing any XML data. Section 3 adapts this algebra to handle XML streams and presents various in-memory, non-blocking evaluation algorithms to process these operators.

2. XML ALGEBRA AND OPTIMIZATION

We are proposing a new algebra and a new algebraic optimization framework well-suited for XML structures and stream processing. In this section, we are presenting the XML algebra without taking into account the fragmentation and reconstruction of XML data. These issues will be addressed in detail in Section 3.

The algebraic bulk operators along with their semantics are given in Figure 1. The inputs and output of each operator are streams, which 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 $\oplus/\{a_1, a_2 \ldots, a_n\} = a_1 \oplus a_2 \oplus \cdots \oplus a_n$, while for a bulk monoid, such as $\cup$, we have $\oplus/\{a_1, a_2 \ldots, a_n\} = \{a_1\} \oplus \{a_2\} \oplus \cdots \oplus \{a_n\}$. Sorting is captured by the special bulk monoid $\oplus$=sort(f), which merges two sorted sequences by f into one sorted sequence by f. For example, sort(f)/$\{2, 1, 3\}$ for $f(x) = -x$ returns $\{3, 2, 1\}$.

The environment, δ, is the current stream record, used in the nested queries. Nested queries are mapped into 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, $v/path$, where v is a stream record attribute and $path$ is a simple XPath of the form: A, $@A$, $path/A$, $path/@A$, $path[n]$, $path/text()$, or $path/data()$, where n is an integer algebraic form. That is, these path forms do not contain wildcard selections, such as $path//A$, and predicate selections, such as $path[e]$. The unnest operation is the only mechanism for traversing an XML tree structure. Function $\mathcal{P}$ is defined over paths as follows:

$$
\begin{array}{rcl}
\mathcal{P}(t, v/path) & = & \mathcal{S}(t.v, path) \\
\mathcal{S}(t, A) & = & \{\ x \mid x \in \text{children}(t),\ \text{tag}(x) = \text{“}A\text{”}\ \} \\
\mathcal{S}(t, path/A) & = & \{\ x \mid v \in \mathcal{S}(t, path),\ x \in \text{children}(v), \\
 & & \qquad \text{tag}(x) = \text{“}A\text{”}\ \} \\
\mathcal{S}(t, path[n]) & = & \{\ \text{children}(v)[n] \mid v \in \mathcal{S}(t, path)\ \}
\end{array}
$$

The extraction operator, ρ, gets an XML data source, T, and returns a singleton stream 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*= $v/A/data()>5$. 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.

Before XQueries are translated to the XML algebra, paths with wildcard selections, such as $e//A$, are instantiated to concrete paths, which may be accomplished, in the absence of type information, with the help of the Tag Structure. Each XPath expression in an XQuery, which may contain wildcard selections, is expanded into a concatenation of concrete XPath expressions. For example, the XPath expression /A//X is expanded to (/A/B/C/X, /A/D/X), if these two concrete paths in the pair are the only valid paths in the Tag Structure that match the wildcard selection.

Our translation scheme from XQuery to the XML algebra consists of two phases: First, XQueries are translated into list comprehensions, which are similar to those in Figure 1. Then, the algebraic forms are derived from the list comprehensions using the definitions in Figure 1.

According to the XQuery semantics [6], the results of nearly all XQuery terms are mapped to sequences of values. This means that, if an XQuery term returns a value other than a sequence, this value is lifted to a singleton sequence that contains this value. A notable exception is a

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

Figure 1: Semantics of the XML Algebra

boolean value, which is mapped to the boolean value itself. The following are few rules for $\mathcal{T}[\![e]\!]$, which maps XQuery terms into algebraic forms. First, we translate XPath terms into simple paths without path predicates. By adopting the semantics of XPath, the implicit reference to the current node in a path predicate, such as from the path A/B in the path predicate X/Y[A/B=1], is captured by the variable $dot, which is bound to the current element. That is, X/Y[A/B=1] is equivalent to X/Y[$dot/A/B=1]. Under this assumption, path predicates are removed in a straightforward way:

$$
\begin{aligned}
\mathcal{T}[\![\$v]\!] &= \{ v \} \\
\mathcal{T}[\![\text{doc}(\text{“url”})]\!] &= \{ \text{doc}(\text{“url”}) \} \\
\mathcal{T}[\![\text{doc}(\text{“url”})/path]\!] &= \{ v \mid d \in \text{doc}(\text{“url”}), \\
&\qquad v \in \mathcal{T}[\![\$d/path]\!] \} \\
\mathcal{T}[\![e_1[e_2]]\!] &= \{ dot \mid dot \in \mathcal{T}[\![e_1]\!], \mathcal{T}[\![e_2]\!] \} \\
\mathcal{T}[\![e[i]]\!] &= \mathcal{T}[\![e]\!][i] \\
\mathcal{T}[\![e[i \textbf{ to } j]]\!] &= \mathcal{T}[\![e]\!][i \textbf{ to } j] \\
\mathcal{T}[\![e/A]\!] &= \mathcal{T}[\![e]\!]/A \\
\mathcal{T}[\![e/@A]\!] &= \mathcal{T}[\![e]\!]/@A
\end{aligned}
$$

where doc is a shorthand for document. To complete our translation scheme, XQuery terms are mapped to the XML algebra:

$$
\begin{aligned}
&\mathcal{T}[\![()]\!] = \{\} \\
&\mathcal{T}[\![e_1, e_2]\!] = \mathcal{T}[\![e_1]\!] +\!\!+ \mathcal{T}[\![e_2]\!] \\
&\mathcal{T}[\![<tag> e </tag>]\!] = \{ \text{element}(\text{“}tag\text{”}, \mathcal{T}[\![e]\!]) \} \\
&\mathcal{T}[\![\textbf{for } \$v \textbf{ in } e_1 \textbf{ where } e_2 \textbf{ return } e_3]\!] \\
&\qquad = \{ w \mid v \in \mathcal{T}[\![e_1]\!], \mathcal{T}[\![e_2]\!], w \in \mathcal{T}[\![e_3]\!] \} \\
&\mathcal{T}[\![\textbf{let } \$v := e_1 \textbf{ return } e_2]\!] = \mathcal{T}[\![e_2[\$v/e_1]]\!] \\
&\mathcal{T}[\![\textbf{some } \$v \textbf{ in } e_1 \textbf{ satisfies } e_2]\!] = \vee/\{ \mathcal{T}[\![e_2]\!] \mid v \in \mathcal{T}[\![e_1]\!] \} \\
&\mathcal{T}[\![e_1 = e_2]\!] = \vee/\{ v_1 = v_2 \mid v_1 \in \mathcal{T}[\![e_1]\!], v_2 \in \mathcal{T}[\![e_2]\!] \}
\end{aligned}
$$

where the function element takes a tag name and a sequence of XML elements and constructs a new XML element, and $e_2[\$v/e_1]$ replaces all free occurrences of variable $\$v$ in the term e_2 with the term e_1. Note that boolean predicates, such as the equality in the last rule, are mapped to one boolean value as is implied by the existential semantics of XQuery predicates [6].

Before the list comprehensions are translated to algebraic forms, the generator domains of the comprehensions are normalized into simple path expressions, when possible. This task is straightforward and has been addressed by earlier work [5, 11]. The following are some examples of normalization rules:

$$
\begin{aligned}
\{ e_1 \mid .\overset{\alpha}{.}., v \in \{ e_2 \mid .\overset{\gamma}{.}. \}, .\overset{\beta}{.}. \} & \\
\qquad \rightarrow \{ e_1 \mid .\overset{\alpha}{.}., .\overset{\gamma}{.}., v \in \{ e_2 \} .\overset{\beta}{.}. \} & \\
\{ e_1 \mid .\overset{\alpha}{.}., v \in \{ e_2 \}, .\overset{\beta}{.}. \} &\rightarrow \{ e_1[v/e_2] \mid .\overset{\alpha}{.}., .\overset{\beta[v/e_2]}{.}. \} \\
\oplus/\{ e \} &\rightarrow e \\
\{ e \}/A &\rightarrow e/A \\
\{ e \mid .\overset{\alpha}{.}. \}/A &\rightarrow \{ e/A \mid .\overset{\alpha}{.}. \}
\end{aligned}
$$

where $.\overset{\beta}{.}.$ indicates a sequence of terms separated by commas while $.\overset{\beta[v/e_2]}{.}.$ changes all terms in the sequence β by replacing all occurrences of v with e_2. The first rule applies when the domain of a comprehension generator is another comprehension and is reduced to a case caught by the second rule. The second rule applies when the domain of a comprehension generator is a singleton sequence. The last rule distributes a tag selection to the header of a comprehension. The fourth rule may look counter-intuitive, but is a consequence of the ambiguous semantics of XQuery where non-sequence values are lifted to singleton sequences. According to the mapping rules and after normalization, a simple path, such as $v/A/B/C, is mapped to itself, that is, to v/A/B/C.

For example, the following XQuery:

```
    for $b in doc("http://www.bn.com")/bib/book
where $b/publisher = "Addison-Wesley"
  and $b/@year >1991
return <book> { $b/title } </book>
```

is translated to the following comprehension using the above mapping rules:

$$
\begin{aligned}
&\{ w \mid b \in \{ v \mid d \in \text{doc}(\text{“http://www.bn.com”}), \\
&\qquad v \in d/\text{bib/book} \}, \\
&\qquad (\vee/\{ v_1 = v_2 \mid v_1 \in b/\text{publisher}, \\
&\qquad\qquad v_2 \in \{\text{“Addison-Wesley”}\} \}) \\
&\qquad \wedge (\vee/\{ v_1 > v_2 \mid v_1 \in b/@\text{year}, v_2 \in \{1991\} \}), \\
&\qquad w \in \{ \text{element}(\text{“book”}, b/\text{title}) \} \}
\end{aligned}
$$

and is normalized into the following comprehension:

$$\{\,\text{element}(\text{“book”}, b/\text{title}) \mid d \in \text{doc}(\text{“http://www.bn.com”}),\ b \in d/\text{bib/book},\ \vee/\{\,v_1 = \text{“Addison-Wesley”} \mid v_1 \in b/\text{publisher}\,\},\ \vee/\{\,v_1 > 1991 \mid v_1 \in b/@\text{year}\,\}\,\}$$

The second phase of our translation scheme maps a normalized term e, which consists of list comprehensions exclusively, into the algebraic form $\mathcal{C}[\![e]\!]\,\{\,\}$, defined by the following rules:

$$\mathcal{C}[\![\{\,e \mid v \in \text{doc}(\text{“url”}),\ .^{\alpha}.\,\}]\!]\,\{\,\} = \mathcal{C}[\![\{\,e \mid\ .^{\alpha}.\,\}]\!]\,(\rho_v(\text{doc}(\text{“url”})))$$

$$\mathcal{C}[\![\{\,e \mid v \in \text{doc}(\text{“url”}),\ .^{\alpha}.\,\}]\!]\,E = \mathcal{C}[\![\{\,e \mid\ .^{\alpha}.\,\}]\!]\,(E \bowtie (\rho_v(\text{doc}(\text{“url”}))))$$

$$\mathcal{C}[\![\{\,e \mid v \in x/path,\ .^{\alpha}.\,\}]\!]\,E = \mathcal{C}[\![\{\,e \mid\ .^{\alpha}.\,\}]\!]\,(\mu^{v,x/path}(E))$$

$$\mathcal{C}[\![\{\,e \mid pred,\ .^{\alpha}.\,\}]\!]\,E = \mathcal{C}[\![\{\,e \mid\ .^{\alpha}.\,\}]\!]\,(\sigma_{\mathcal{C}[\![pred]\!]\,\{\,\}}(E))$$

$$\mathcal{C}[\![\{\,e \mid\ \}]\!]\,E = \Delta^{\cup,e}(E)$$

$$\mathcal{C}[\![\oplus/\{\,e \mid\ .^{\alpha}.\,\}]\!]\,E = \begin{cases} \oplus/(\mathcal{C}[\![\{\,e \mid\ .^{\alpha}.\,\}]\!]\,E) \\ \text{where } \oplus/(\Delta^{\cup,e}(e')) = \Delta^{\oplus,e}(e') \end{cases}$$

In addition, since the predicates used in our XML algebraic operators have existential semantics, similar to the semantics of XQuery predicates, we move the existentially quantified variables ranged over paths into the predicate iself, making it an implicit existential quantification. This is achieved by the following rules:

$$\begin{aligned} \mathcal{C}[\![\vee/\{\,e \mid\ .^{\alpha}.,\ v \in path,\ .^{\beta}.\,\}]\!] &= \mathcal{C}[\![\vee/\{\,e[v/path] \mid\ .^{\alpha}.,\ .^{\beta}.\,\}]\!] \\ \mathcal{C}[\![\vee/\{\,e \mid\ \}]\!] &= \mathcal{C}[\![e]\!] \end{aligned}$$

For example, $\vee/\{\,v_1 = \text{“Addison-Wesley”} \mid v_1 \in b/\text{publisher}\,\}$ is reduced to $b/\text{publisher} = \text{“Addison-Wesley”}$, which has implicit existential semantics.

Under these rules, the normalized list comprehension derived from our example query is mapped to:

$$\Delta^{\cup,h}(\sigma_{p_2}(\sigma_{p_1}(\mu^{b,\text{d/bib/book}}(\rho_d(\text{document}(\text{“http://www.bn.com”})))))$$

where

$$\begin{aligned} h &= \text{element}(\text{“book”}, b/\text{title}) \\ p_1 &= (b/\text{publisher} = \text{“Addison-Wesley”}) \\ p_2 &= b/@\text{year} > 1991 \end{aligned}$$

There are many algebraic transformation rules that can be used in optimizing the XML algebra. Some of them have already been used successfully for the relational algebra, such as evaluating selections as early as possible. We are concentrating here on query unnesting (query decorrelation) because it is very important for processing streamed data. Without query unnesting, nested queries must be evaluated in a nested-loop fashion, which requires multiple passes through the stream of the inner query. This is unacceptable because of the performance requirements of stream processing.

Our unnesting algorithm is shown in Figure 2.A: for each box, q, that corresponds to a nested query, it converts the reduction on top of q into a nest, and the blocking joins/unnests that lay on the input-output path of the box q into outer-joins/outer-unnests in the box q' (as is shown in the example of Figure 2.B). At the same time, it embeds the resulting box q' at the point immediately before is used. There is a very simple explanation why this algorithm is correct: The nested query, q, in Figure 2.A, consumes the same input stream as that of the embedding operation, opr, and computes a value that is used in the component, f, of the embedding query. If we want to splice this box onto the stream of the embedding query we need to guarantee two things. First, q should not block the input stream by removing tuples from the stream. This condition is achieved by converting the blocking joins into outer-joins and the blocking unnests into outer-unnests (box q'). Second, we need to extend the stream with the new value v of q before it is used in f. This manipulation can be done by converting the reduction on top of q into a nest, since the main difference between nest and reduce is that, while the reduce returns a value (a reduction of a stream of values), nest embeds this value to the input stream. At the same time, the nest operator will convert null values to zeros so that the stream that comes from the output of the spliced box q' will be exactly the same as it was before the splice.

We now apply our translation and optimization scheme over the following XQuery (taken from [6]):

```
<result>{
    for $u in document("users.xml")//user_tuple
    return
      <user>
        { $u/name }
        { for $b in document("bids.xml")
                    //bid_tuple[userid = $u/userid]/itemno,
              $i in document("items.xml")
                    //item_tuple[itemno = $b]
          return <bid>{ $i/description/text() }</bid>
          sortby(.) }
      </user>
    sortby(name) } </result>
```

which lists all users in alphabetic order by name so that, for each user, it includes descriptions of all the items (if any) that were bid on by that user, in alphabetic order. Recall that sorting is captured in our algebra using the monoid sort(f), which sorts a sequence by f. The algebraic form of the above query is:

$$\Delta^{\text{sort}(u/\text{name}),h_1}(\mu^{u,d_a/\text{users/user\_tuple}}(\rho_{d_a}(\text{document}(\text{“users.xml”}))))$$

where the header h_1 is the XML construction:

$$\text{element}(\text{“user”}, u/\text{name} +\!\!+ h_2)$$

and h_2 contains a nested algebraic form:

$$\begin{aligned} h_2 &= \Delta^{\text{sort}(.),\text{element}(\text{“bid”},i/\text{description/text}())}(X \bowtie_{i/\text{itemno}=b} Y) \\ X &= \mu^{b,b_t/\text{itemno}}(\mu^{b_t,d_b/\text{bids/bid\_tuple}}_{b_t/\text{userid}=u/\text{userid}}(\rho_{d_b}(\text{doc}(\text{“bids.xml”})))) \\ Y &= \mu^{i,d_i/\text{items/item\_tuple}}(\rho_{d_i}(\text{doc}(\text{“items.xml”}))) \end{aligned}$$

After query unnesting, the resulting decorrelated query is:

$$\Delta^{\text{sort}(u/\text{name}),\text{element}(\text{“user”},u/\text{name} +\!\!+ x)}(\Gamma^{x,\text{sort}(.),\text{element}(\text{“bid”},i/\text{description/text}())}_{d_a}(X))$$

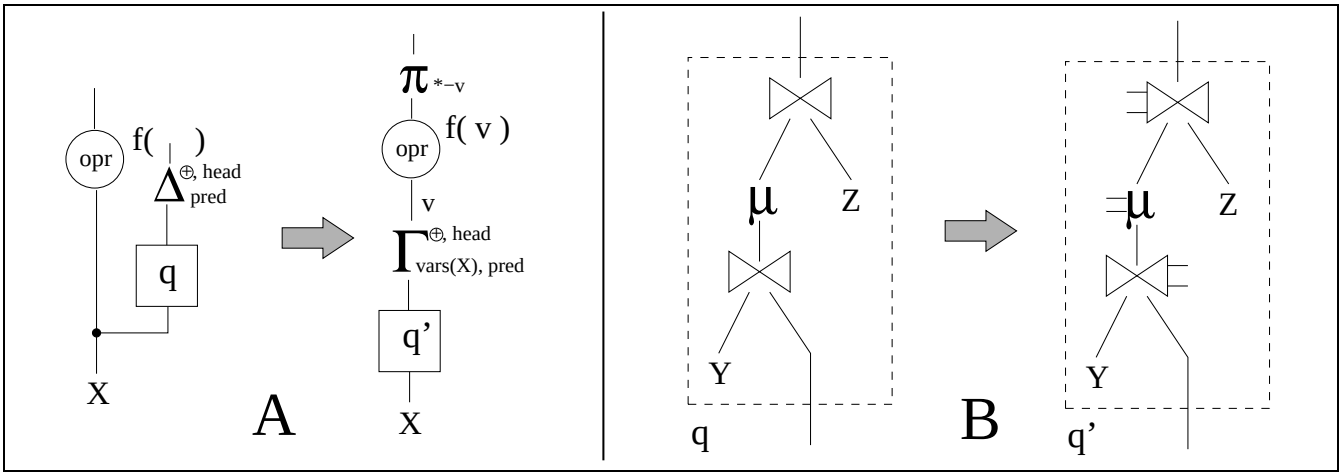

Figure 2: Algebraic Query Unnesting

where

$$
\begin{aligned}
X &= (=\!\!\not\mu^{b,b_t/\text{itemno}}(Y)) =\!\!\bowtie_{i/\text{itemno}=b} Z \\
Y &= A =\!\!\bowtie_{b_t/\text{userid}=u/\text{userid}} B \\
A &= \mu^{u,d_a/\text{users/user\_tuple}}(\rho_{d_a}(\text{doc(“users.xml”)})) \\
B &= \mu^{b_t,d_b/\text{bids/bid\_tuple}}(\rho_{d_b}(\text{doc(“bids.xml”)})) \\
Z &= \mu^{i,d_i/\text{items/item\_tuple}}(\rho_{d_i}(\text{doc(“items.xml”)}))
\end{aligned}
$$

where $=\!\!\bowtie$ and $=\!\!\not\mu$ are left-outer join and outer unnest, respectively.

3. PROCESSING CONTINUOUS STREAMS

In our framework, streamed XML data are disseminated to clients in the form:

<stream:xxx hid="...">XML fragment </stream:xxx>

where xxx is the tag name filler, repeat, replace, or remove. The hole ID, hid, specifies the location of the fragment in the reconstructed XML tree. Our framework uses the XML algebra for processing XML streams. The main modification needed to the algebra is related to the evaluation of the simple path expressions used in predicates and the unnest operator since now they have to be evaluated against the document fillers. More specifically, function $\mathcal{P}$, given in Section 2, must now be extended to handle holes:

$$\mathcal{S}(\text{<stream:hole hid = “}n\text{”/>}, path) \quad = \quad \perp$$

The returned bottom value, $\perp$, indicates that the path has not been completely evaluated against the current XML fragment due to a hole in the fragment. If a $\perp$ is returned at any point during the path evaluation (as is defined by $\mathcal{P}$), the XML fragment is suspended. Each client has one central repository of suspended XML fragments (regardless of the number of input streams), called *fragment repository*, which can be indexed by the combination of the stream number and hid (the hid of the stored filler). In addition, each operator has a local repository that maps hole IDs to filler IDs in the fragment repository. Suppose, for example, that a filler with hid=m is streamed through an XML algebraic operator and that this operator cannot complete the evaluation of three paths due to the holes with hid's h_1, h_2, and h_3. Then, the local repository of the operator will be extended with three mappings: (h_1, m), (h_2, m), and (h_3, m). When later the filler for one of the holes arrives at the operator, say the filler for h_2, then the hole h_2 inside the filler with hid=m in the fragment repository is replaced with the newly arrived filler. The h_2 mapping is removed from the local repository of the operator and the resulting filler with hid= m at the fragment repository is evaluated again for this operator, which may result to more hole mappings in the local repository of the operator. Finally, if there are no blocking holes during path evaluation, that is, when operator paths are completely evaluated, then the filler is processed by the operator and is removed from the fragment repository. It is not necessary for all the holes blocking the evaluation of the operator paths to be filled before the fragment is processed and released. For example, if both paths in the selection predicate $v/A/B = 3 \wedge v/C/D > 4$, are blocked by holes, then if one of the holes is filled and the corresponding predicate is false, there is no need to wait for the second hole to be filled since the fragment can be removed from the stream.

If the server sends repeat, replace, or remove fragments, then the client modifies its fragment repository and may stream some of the fragments through the algebraic operators. If it is a repeat fragment and its hid is already in the fragment repository, then it is ignored, otherwise it is streamed through the query as a filler fragment (since the client may have been connected in the middle of the stream broadcast). If it is a replace fragment and its hid is already in the fragment repository, then the stored fragment is replaced, otherwise it is streamed through the query as a filler fragment. Finally, if it is a remove fragment, then the fragment with the referenced hid is removed from the fragment repository (if it exists).

Our evaluation algorithms for the XML algebra are based on in-memory hashing. Blocking operators, such as join and nest, require buffers for caching stream records. The evaluation algorithms for these operators are hash-based, when possible (it is not possible for non-equijoins). For example, for the join $X \bowtie_{x/A/B=y/C/D/E} Y$, we will have two hash tables in memory with the same number of buckets, one for stream X with a hash function based on the path $x/A/B$ and one for the stream Y based on the path $y/C/D/E$. When both streams are completed, the join is evaluated by joining the associated buckets in pairs. The nest operator

can be evaluated using a hash-based algorithm also, where the hash key is the combination of all group-by paths.

As we have mentioned, to cope with continuous streams and with streams larger than the available memory, we make use of punctuations sent by the server along with the data. In our framework, when punctuations are received at the client site, they are stored at a central repository, called *punctuation repository*, and are indexed by their stream number. Each blocking operation, such as join or nesting, is associated with one (for unary) or two (for binary) pairs of stream numbers/hash keys. The hash key is a path that can be identified by a Tag Structure ID. The blocking operators are evaluated continuously by reading fragments from their input streams without producing any output but by filling their hash tables. As soon as one of their hash tables is full, they consult the punctuation repository to find all punctuations that match both their stream number and Tag Structure ID. Then they perform the blocking operation over those hashed fragments that match the retrieved punctuations, which may result to the production of some output. The last phase, flushing the hash tables, can be performed in a pipeline fashion, that is, one output fragment at a time (when is requested by the parent operation). The punctuation repository is cleared from all the punctuation of a stream when the entire stream is repeated (i.e., when the root fragment is repeated).

Sorting is a special case and is handled separately. It is implemented with an in-memory sorting, such as quicksort. Each sort operator remembers the largest key value of all data already flushed from memory. In the beginning, of course, the largest key is null. Like the other blocking operators, when the buffer used for sorting is full, the punctuation repository is consulted to find punctuations related to the sorting key. If there is a punctuation that indicates that we have seen all data between the largest key value (if not null) and some other key value, then all buffered data smaller or equal to the latter key value are flushed and this key value becomes the new largest key.

4. CONCLUSION

We have presented a framework for processing streamed XML data based on an XML algebra and an algebraic optimization framework. The effectiveness of our framework depends not only on the available resources at the client site, especially buffer size, and on the ability of the client to optimize and evaluate queries effectively, but also on the willingness and the ability of servers to broadcast useful punctuations through the data stream to help clients utilize their resources better. The server must be aware of all possible anticipated client queries and disseminate punctuations that reduce the maximum and average sizes of client resources. In addition, the server can disseminate the fragmentation/repetition policy used in splitting its XML data as well as statistics about the data sent between punctuations before streaming the actual data. This information may help clients to allocate their limited memory to various query operations more wisely. We are planning to address these issues in future work.

5. REFERENCES

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast Disks: Data Management for Asymmetric Communications Environments. In *ACM SIGMOD International Conference on Management of Data, San Jose, California*, pages 199–210, May 1995.

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

[3] C. Beeri and Y. Tzaban. SAL: An Algebra for Semistructured Data and XML. In *ACM SIGMOD Workshop on The Web and Databases (WebDB'99), Philadelphia, Pennsylvania*, pages 37–42, June 1999.

[4] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *ACM SIGMOD International Conference on Management of Data, Montreal, Canada*, pages 505–516, May 1996.

[5] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

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

[7] V. Christophides, S. Cluet, and J. Siméon. On Wrapping Query Languages and Efficient XML Integration. In *ACM SIGMOD International Conference on Management of Data, Dallas, Texas*, pages 141–152, May 2000.

[8] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your Mediators Need Data Conversion! In *ACM SIGMOD International Conference on Management of Data, Seattle, Washington*, pages 177–188, June 1998.

[9] S. Cluet and G. Moerkotte. Nested Queries in Object Bases. In *Workshop on Database Programming Languages, Gubbio, Italy*, September 1995.

[10] L. Fegaras and R. Elmasri. Query Engines for Web-Accessible XML Data. In *VLDB Conference, Roma, Italy*, pages 251–260, 2001.

[11] L. Fegaras and D. Maier. Optimizing Object Queries Using an Effective Calculus. *ACM Transactions on Database Systems*, 25(4):457–516, December 2000.

[12] M. Fernandez, J. Simeon, and P. Wadler. An Algebra for XML Query. In *FST TCS, Delhi*, December 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] J. Hellerstein, P. Haas, and H. Wang. Online Aggregation. In *ACM SIGMOD International Conference on Management of Data, Tucson, Arizona*, pages 171–182, May 1997.

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

[16] A. Wilschut and P. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *First International Conference on Parallel and Distributed Information Systems, Miami Beach, Florida*, pages 68–77, December 1991.

[17] World Wide Web Consortium (W3C). *Extensible Markup Language (XML)*. http://www.w3.org/XML/.