XFrag: A Query Processing Framework for Fragmented XML Data

Sujoe Bose and Leonidas Fegaras
University of Texas at Arlington, CSE
416 Yates Street, P.O. Box 19015
Arlington, TX 76019-19015
{bose,fegaras}@cse.uta.edu

ABSTRACT

Data fragmentation offers various attractive alternatives to organizing and managing data, and presents interesting characteristics that may be exploited for efficient processing. XML, being inherently hierarchical and semi-structured, is an ideal candidate to reap the benefits offered by data fragmentation. However, fragmenting XML data and handling queries on fragmented XML are fraught with challenges: seamless XML fragmentation and processing models are required for deft handling of query execution on inter-connected and inter-related XML fragments, without the need of reconstructing the entire document in memory. Recent research has studied some of the challenges and has provided some insight on the data representation, and on the rather intuitive approaches for processing fragmented XML. In this paper, we provide a novel pipelined framework, called *XFrag*, for processing XQueries on XML fragments to achieve processing and memory efficiency. Moreover, we show that this model is suitable for low-bandwidth mobile environments by accounting for their intrinsic idiosyncrasies, without sacrificing accuracy and efficiency. We provide experimental results showing the memory savings achieved by our framework using the XMark benchmark.

1. INTRODUCTION

The widespread adoption of XML has rendered it as the language of choice for communication between collaborating systems. XML is also being studied as a data storage format and is being used by various systems for data management and query processing on native XML data [1, 2]. XQuery has become the language of choice for querying stored XML data, but recently we have seen systems, such as XSM [25], XRQL [4], and FluXQuery [3], that support XQuery processing on streamed XML data as well. XML data, being inherently hierarchical and semi-structured, poses an overwhelming overhead on critical runtime factors, such as memory requirements and processing efficiency. Given that most of queries on large XML documents are selective in nature, queries may benefit from fragmenting the XML document so that processing in parts would require less memory and processing power. There are several other reasons for fragmenting data. In a realtime sensor-based system, data is continuously generated from sensors and it disseminated in fragments as and when it occurs. Furthermore, streaming changes to data may pose less overhead by sending only fragments corresponding to the change, rather than sending the entire document with the change. Moreover, given the current shift from pull-based to push-based broadcast models, fragmentation of data provides several benefits: it is possible to prioritize data fragments so that high priority fragments of data may be scheduled ahead of the low priority ones. Also it is possible to associate quality of service parameters on the data fragments to meet delivery constraints. With the proliferation of mobile devices and with the quest for information on the move, servers disseminate data over low-bandwidth and error-prone environments. As the intermittent connectivity of mobile clients makes infeasible to deliver huge datasets, fragments may be a better choice for data delivery. Moreover it is easier to synchronize on smaller fragments because transmitting changes to data requires only sending the fragments that correspond to the change, without having to send the entire document.

The hallmark of our framework is the support for processing fragments rather than documents, especially in the presence of continuous updates to the document. This helps in optimizing the bandwidth and processing requirements by transmitting and processing only the update fragments without its entire unchanged context. Another area that benefits from a fragmented XML data model, is the inclusion of temporal extents on the XML dataset to capture the historical context of the transmitted data. In the new breed of event driven applications, as presented in XCQL [23], which require implicit temporal association, transmission in fragments provides seamless integration of temporal context within the data model and constructs for performing historical and window queries with temporal extents. In data distribution systems, fragmentation of data items is prevalent, because collaborating systems, geographically and logically dispersed, require different aspects of the data. Furthermore, system efficiency is facilitated by useful work in processing the required data and by ignoring the rest.

Unfortunately, processing fragments instead of whole XML documents is fraught with challenges. It requires not only knowledge of the locational context of fragments, which al-

 Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland.

lows us to navigate from fragment to fragment during query processing, but also caching some of the fragments when necessary, since not all fragments may be available at the same time. Also, due to changes to fragments and to the intermittent connectivity of mobile clients, fragments may arrive in any order, and may be repeated or updated. In this paper, we address these challenges and provide a robust framework to process the fragments as and when they arrive without losing the overall context, resulting in lower memory footprints and faster response time.

Our previous work [18, 16] has concentrated on modeling and management of fragmented XML data and has proposed simple methods to handle the challenges present in such representation. One common way is to suspend fragments until their contained data arrives for continuing execution. This causes a serious challenge on the memory requirements, as fragments may arrive in any order. In addition, waiting for a fragment to come with complete information necessary for execution would result in blocking.

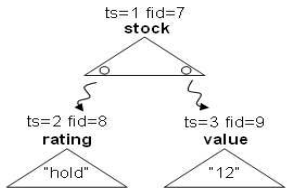
In this paper, we propose a novel pipelined execution framework for processing XQueries on streamed XML fragments. The fragments are processed as and when they arrive, and their inter-dependencies and hence their effect on the query results are resolved pro-actively. Our motivation to process the fragments as soon as possible is to conserve memory by discarding fragments that will not contribute to the result. Moreover, fragments that do not actively contribute to the result, but due to their relationship with other fragments affect the result, as in the case of fragments involved in query predicates, are kept in memory as long as necessary.

Unlike traditional applications of the pipelined processing model, query processing of fragmented data using the pipelined model of execution provides new challenges: since queries could span across fragments, we must factor the relative references between fragments while executing the query predicates and projections. Additionally, queries on XML data could operate on any level of the XML document, and hence the query predicates and projections traverse multiple fragments, which may arrive at arbitrary times in the fragmented XML stream. Also, the ability to construct new elements as part of the result XML, the presence of accumulation operators, and the out of order arrival of XML fragments, add additional challenges to the processing framework. Note that, we assume that the query clients, such as low-power hand-held devices, have limited memory and processing capacity that make it impossible to reconstruct the entire XML data before processing the queries.

The rest of the paper is organized as follows. Section 2 presents the related work in the area of XML stream query processing. Section 3 explains our framework by providing a model for XML fragments and for tag structures, which define the structural makeup of fragments. Section 4 describes in detail the pipeline model of processing fragments and the formal representation of the translation and processing framework used in XFrag. Finally, Section 6 presents experimental results from our implementation and shows the memory saving achieved in our framework.

2. RELATED WORK

Several recent efforts have focused on addressing frameworks for continuous processing of data streams [5, 8, 9, 21], however to the best of our knowledge, there is no work done



Query: stream(stock)/stock[rating ≥ "hold"]/value

Figure 1: Sample Stock Fragments

in stream query processing of fragmented XML data. The Tribeca [13] data stream processing system provides language constructs to perform aggregation operations, as well as multiplexing and window constructs to perform stream synchronization, but it is restricted to relational data. Other efforts concentrating on windowed stream processing, such as StreamQuel [20], CQL [17], also address relational data only and provide SQL-like constructs to process streaming data. The COUGAR [11] system proposes the use of ADTs in object-relational database systems to model streams with associated functions to perform operations on the stream data. Several efforts have addressed the stream processing of XML data using XPath expressions [5, 7, 9]. A transducer-based XQuery processor for streaming XML data has been proposed in [25]. An alternative to transducer-based processing is a compositional XQuery processor based on SAX events, defined in [15]. An alternative fragmented XML processing model, suitable for pull-based web-service applications, is presented in Active XML [26]. In Xstream [24], the advantages of a semantics-based fragmentation of XML data for efficient transmission over a wireless medium are highlighted.

3. OUR FRAGMENTED DATA MODEL

In our framework, the basic stream components transmitted by a server are fragments, each with its own ID. To be able to relate fragments with each other, we derive the concept of *holes* and *fillers* as detailed in our earlier work [18]. A hole represents a placeholder into which another rooted subtree (a fragment), 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, and so on. An example set of XML fragments is shown in figure 1.

Our framework makes use of the structural summary of XML data, called the *Tag Structure*, which defines the structure of data and provides information about fragmentation. This information is used when compiling XQuery expressions into plans that operate on the XML fragments and when deciding which fragments to keep in memory. Moreover, the Tag Structure is used in expanding wild-card path selections in queries to optimize query execution. The Tag Structure is itself structurally a valid XML fragment that conforms to the following simple recursive DTD:

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

A tag corresponds to an XML tagged element and is qualified by a unique id, a name (the element tagname), and

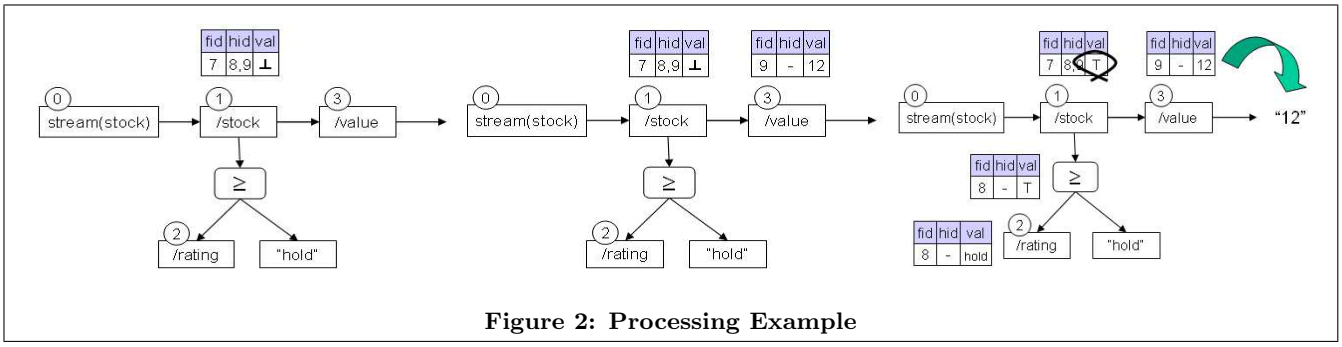


Figure 2: Processing Example

a type. A filler type implies that this element will arrive in a separate filler fragment, as opposed to the embedded type, which implies that this element is embedded within its parent element (inside the same fragment). Since the tag structure is a very important piece of information to the client for handling the input stream data, we require that it be streamed before the actual data.

4. THE XFRAG PIPELINE

Each XQuery primitive in an input query corresponds to an XFrag operator, which operates on an XML subtree at a particular level in the original XML document. For example, a path step in an XPath expression corresponds to a path operator that operates on the elements in the XML document having the same tag value and it corresponds to the same subtree level as that of the operator. Moreover, an XPath predicate expression maps to a condition operator, which may in turn reference path operators to perform predicate evaluation. As each fragment corresponds to a particular level in the original XML document, it is necessary to associate each operator with the fragments that will process. We use the Tag Structure to associate operators with the tag structure id (tsid) of the subtree that corresponds to the execution context of the operator. Each fragment is identified by the tsid of the subtree that belongs to in the original XML document, and hence each operator will process the fragment only if the tsids match. In the event that they do not match, the fragment is simply passed on to the subsequent operator in the query tree.

4.1 Fragment Relationships

As fragments in the original document may arrive in any order and query expressions may contain predicates at any level in the XML tree, it is necessary to keep track of the parent-child links between the various fragments, so that if a particular fragment does not pass the predicate evaluation at a particular level in the XML document, the corresponding descendant fragments must not be rendered as part of the output. We use the filler-id and hole-id information in the fragments to keep track of the fragment relationships. We maintain the fragment links in an association table at each operator to record the parent-child relationships seen in fragments processed by the operator. Moreover, each entry is tagged by a value of true, false, undecided ($\perp$), or a result fragment. While the former three values are possible in intermediate operators that do not produce a result, the latter is possible when the operator is the terminal operator in the query tree branch. Fragments corresponding to intermediate operators are discarded after recording the parent-child link relationships, thereby conserving memory. This link information corresponds to a small part of the actual data in the XML fragment, the rest of which is not relevant in producing the result.

4.2 Ancestor Inquiry

When a fragment is processed by an operator, it needs to verify if the predecessor operator has excluded its parent fragment due to either predicate failure or due to exclusion of its ancestor. For this reason, each operator maintains both a successor operator list and a pointer to the predecessor operator, using the former to hand-over fragments for processing by successor operators, and the latter to resolve fragment relationships and predicate criteria. When the predecessor inquiry is made to determine the eligibility of a particular fragment, one of four conditions may arise. (1) The parent fragment may not have arrived at the predecessor and hence there is no entry in the association table of the predecessor. In this case, the fragment is tagged with an undecided value. (2) The parent fragment had arrived at the predecessor and is tagged with an undecided value. In this case too the fragment with an undecided value is recorded. (3) The parent fragment had arrived at the predecessor and is tagged with a value of true, which implies that the parent fragment or its ancestor have passed all predicate expressions. In this case the fragment is tagged with a value of true, it is a potential candidate for output, depending on whether this operator is the result producer or is an intermediary in the query tree. (Note that the predicate evaluation follows the existential semantics of XQuery.) The last case is when the parent fragment had arrived and is tagged with a value of false. In this case, the fragment is also tagged with the value of false.

4.3 Descendant Trigger

As fragments may be waiting on operators to decide on their ancestor eligibility, they must be triggered when an ancestor condition is evaluated. Moreover, the predicate evaluation of a fragment may depend on child tags embedded in the fragment or child fragments that may arrive at a later point in time. In order to account for these dependencies, we introduce a recursive trigger invocation on the successor operators. Whenever a fragment is marked as true (or false) in a particular operator, other fragments that are waiting with an undecided value in successor operators may now be triggered to be rendered with a value of true (or false), and, subsequently, either produced (or not produced) as output. Similarly, condition operators will trigger their parent operators when a condition evaluates to true for at least one of

$$
\begin{aligned}
\mathcal{R}([\![\text{stream}(url)\,path]\!]_\delta,\beta,ts) &\rightarrow \{(\beta',ts') \mid ts' \leftarrow \mathbf{ts}(url), \beta' \leftarrow \rho^{url}_{s,p,\omega,t}(t = ts'/@tsid, s = \mathcal{R}([\![path]\!]_\delta,\beta',ts')\}\\
\mathcal{R}([\![/A\,path]\!]_\delta,\beta,ts) &\rightarrow \{(\beta',ts') \mid ts' \leftarrow ts/tag[@name = \text{“}A\text{”}],\\
&\qquad \beta' \leftarrow \mu^{A}_{s,p,\omega,t}(t = (\mathsf{isfiller}(ts')?ts'/@tsid : \beta.tsid), p = \beta,\\
&\qquad\qquad s = \mathcal{R}([\![path]\!]_\delta,\beta',ts'))\}\\
\mathcal{R}([\![/@A]\!]_\delta,\beta,ts) &\rightarrow \{\ (\beta',ts) \mid \beta' \leftarrow \mu^{@A}_{s,p,\omega,t}(t = ts.tsid, p = \beta, s = \{\})\ \}\\
\mathcal{R}([\![/*\,path]\!]_\delta,\beta,ts) &\rightarrow \{\ (\beta',ts') \mid ts' \leftarrow ts/tag, \beta' \leftarrow \mu^{ts'/@name}_{s,p,\omega,t}(t = (\mathsf{isfiller}(t')?t'.tsid : \beta.tsid),\\
&\qquad\qquad p = \beta, s = \mathcal{R}([\![path]\!]_\delta,\beta',ts'))\}\\
\mathcal{R}([\![//A\,path]\!]_\delta,\beta,ts) &\rightarrow \mathcal{R}([\![/A\,path]\!]_\delta,\beta,ts) \cup \mathcal{R}([\![/*\,//A\,path]\!]_\delta,\beta,ts)\\
\mathcal{R}([\![/A[e]\,path]\!]_\delta,\beta,ts) &\rightarrow \{\ (\beta',ts') \mid ts' \leftarrow ts/tag[@name = \text{“}A\text{”}],\\
&\qquad \beta' \leftarrow \mu^{A,c}_{s,p,\omega,t}(t = (\mathsf{isfiller}(t')?t'.tsid : \beta.tsid), p = \beta,\\
&\qquad\qquad s = \mathcal{R}([\![path]\!]_\delta,\beta',ts'), c = \mathcal{R}([\![e]\!]_\delta,\beta',ts'))\}\\
\mathcal{R}([\![e_1\ \mathsf{op}\ e_2]\!]_\delta,\beta,ts) &\rightarrow \{\ (\beta',ts) \mid \beta' \leftarrow \sigma^{lhs,rhs,op}_{s,p,\omega,t}(t = ts/@tsid, p = \beta, s = \{\},\\
&\qquad\qquad lhs = \mathcal{R}([\![e_1]\!]_\delta,\beta,ts), rhs = \mathcal{R}([\![e_2]\!]_\delta,\beta,ts))\ \}\\
\mathcal{R}([\![\text{“}text\text{”}]\!]_\delta,\beta,ts) &\rightarrow \{\ (\beta',ts) \mid \beta' \leftarrow \mathcal{C}^{text}_{s,p,\omega,t}(t = ts, p = \beta, s = \{\})\ \}\\
\mathcal{R}([\![<A>e</A>]\!]_\delta,\beta,ts) &\rightarrow \{\ (\beta',ts') \mid id \leftarrow \mathsf{genTsid}(), ts' \leftarrow \ \text{<tag name=“A” id=“}id\text{”>}ts\text{</tag>},\\
&\qquad \beta' \leftarrow \theta^{A}_{s,p,\omega,t}(t = ts'/@tsid, p = \beta, s = \mathcal{R}([\![path]\!]_\delta,\beta',ts'))\ \}
\end{aligned}
$$

Figure 3: XFrag Query Translation

the child tags or fragments.

4.4 XFrag Pipeline Processing Example

As an example, consider the stock stream, which produces fragments corresponding to values and ratings of stocks, with sample fragments as shown in Figure 1. The stock stream is described by the following tag structure:

```
<tag stream="stock">
   <tag id="1" name="stock" filler="true">
      <tag id="2" name="rating" filler="true"/>
      <tag id="4" name="symbol"/>
      <tag id="5" name="name"/>
      <tag id="3" name="value" filler="true"/>
   </tag> </tag>

Query 1: stream("stock.xml")
           /stock[rating >= "hold"]/value
```

The XFrag pipeline corresponding to the above query is depicted in Figure 2. When a “stock” fragment with tsid 1, filler id 7 and hole ids 8 and 9, arrives at the operator with tsid 1, the association table is updated with this information as shown in figure 2(a). Moreover, the fragment 7 is tagged with an undecided value, as the condition has not been evaluated yet for this fragment. Note that, at this point, the “stock” filler may be discarded as it is no more needed to produce the result and the hole filler association is already captured. This results in memory conservation on the fly, as we discard fragments, if they are no more needed to be retained. When the “value” fragment corresponding to the “stock” filler arrives, the operator with tsid 3 updates its association table with the value of its expression, but does not output the value, as the inquiry on the predecessor operator returns an undecided value. The “value” filler may also be discarded at that point conserving memory, as the result value, which is a subset of the fragment, is already captured in the association table. When the “rating” fragment corresponding to the “stock” filler arrives, the operator with tsid 2 updates its association table and returns the value of “hold”, as there is no condition for it to wait. The condition operator now determines that this value matches the criteria for filler id 8 and hence triggers the parent “stock” operator with the id 8 as true. The “stock” operator updates its association table for the parent filler 7 as true and triggers its successor “value” operator, which causes the value of “12” to be output.

5. XFRAG FORMAL SEMANTICS

5.1 Query Translation Function

The translation of XQuery expressions into the XFrag operator pipeline is depicted in Figure 3. The translation function $\mathcal{R}$, is a mapping from XQuery expression and the tag structure to an XFrag operator tree. Every operator is a specialization of the basic operator type β, which is characterized by a successor operator list s, a predecessor p, an association table ω, and the tag structure corresponding to the operator. The stream extraction operator ρ reads fragments from a stream, identified by url, and forwards them to the successors. Path expressions are mapped to the path projection operator μ. Wild-card and descendant path expressions are translated into a set of path projection operators by performing a wild-card projection and recursive descent on the tag structure. Predicate expressions are translated into condition operators and element construction into the construction operator θ. Since element construction adds a new tag element into the result set, the tag structure is extended with a tag equal to the element tag and a new tsid generated to identify the tag. A FLWR expression, which binds an expression to a variable, extends the environment δ, with a binding entry that relates the variable name to the XFrag operator sub-tree corresponding to the bound expression. The bindings added in the environment are referenced at the point where a variable is used in other expressions. Using the translation rules, the query used in the stock example is converted into the XFrag operator tree:

$$\rho^{stock.xml}_{s,p,\omega,t0}(\mu^{stock}_{s,p,\omega,t1}(\sigma(\geq, \mu^{rating}_{s,p,\omega,t2}, c^{hold}), \mu^{value}_{s,p,\omega,t3}))$$

5.2 Fragment Processing Function

The semantics of fragment handling by the various operators in XFrag is shown in Figure 4. For brevity, we have not included the semantics for all the operators, but have presented those that are used in our example. There are three basic functions defined for each operator in XFrag. The process function $\mathcal{P}$ takes a fragment and produces a set of output fragments. The inquiry function $\mathcal{I}$ takes a filler id and returns the value recorded in the association table of the operator and in any conditional expression, if present. The trigger function $\mathcal{T}$ takes a filler id and returns a set of fragments as output. The process function performs an inquiry on the association table, and, depending on the result of the inquiry and on whether it is an intermediate operator, triggers successors to output any fragments waiting to be resolved. The operators corresponding to the FLWR expressions, not presented, requires special mention. While the operator corresponding to the "FOR" expression produces result fragments as and when a fragment is available on the return clause, the "LET" expression, on the other hand, collects fragments from the return clause until all the siblings are present and then produces the result.

6. EXPERIMENTAL RESULTS

We have implemented the XFrag framework in Java and have modeled the operator types as individual classes. All operators are made to derive from the common fragment operator that provides the basic components of the XFrag pipeline operator and the supporting functions. We have ran tests using the XMark benchmark [22] on a Pentium III processor running Microsoft Windows 2000 with 512MB RAM. We have used the following 3 queries on the generated auction XML document and compared the results with the Qizx XQuery processor [10].

```
Query 1: doc("auction.xml")/site/open_auctions//
           increase
Query 2: doc("auction.xml")/site/open_auctions/
           open_auction[initial > "10"]/bidder
Query 3: doc("auction.xml")/site/open_auctions/
           open_auction/bidder[increase > "200"]
```

The memory profiling was done using the EclipseProfiler plugin for the Eclipse IDE and the results are summarized in Figure 5, using a generated auction XML document of size 23.3MB. For XFrag, the auction document was fragmented into fillers and holes, producing a file of size 27.3MB, and the resulting filler fragments were processed sequentially. Note that the running time for XFrag was about twice as much as for Qizx, however, our main focus was the memory consumption to suit processing using devices with less memory. While the Qizx XQuery processor took almost the same amount of memory to run all of the three queries, topping about 60MB of heap space usage, the XFrag framework took a maximum of 10MB of heap space. Moreover, for the first Query, it took a constant amount of memory of about 2MB, as there were no conditional expressions to be evaluated and hence fragments were output as they arrived without waiting on other related fragments. For Queries 2 and 3, the association tables were populated with the hole filler links,

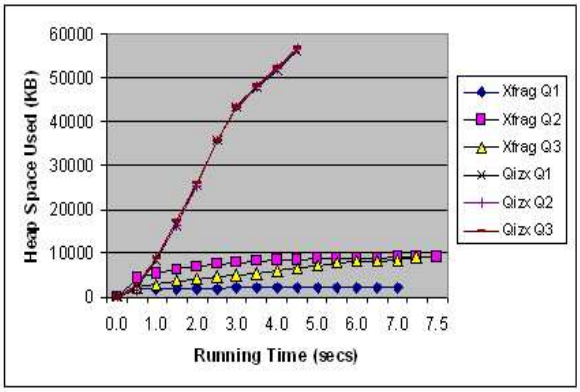


Figure 5: Comparison of Heap space usage for `XFrag` and `Qizx`

and the fragment values suspended until a matching condition signals the output to be flushed. While Query 2 had an initial increase compared to Query 3, which had a smoother increase, they both consumed the same amount of memory towards the end of the stream. Since Query 2 had to keep track of the filler-hole links from the "open_auction" fillers to the corresponding "bidder" fragments that occur later in the fragmented XML auction data, the memory consumption increased initially and then sustained when the "bidder" fragments arrived producing continuous output. However, Query 3 did not have to maintain these links as there is no condition expression present except in the "bidder" fragments only. In both Query 2 and 3, the overhead in memory consumption is due to the growth in the association table entries, however the huge advantage gained in the aggressive flushing of fragments eclipses the association table overhead.

7. CONCLUSION AND FUTURE PLANS

We have presented the XFrag framework to process fragments of XML data without having to wait for the entire XML document to be received and materialized. The fragments are processed as and when they occur and any interdependencies are pro-actively resolved, resulting in memory conservation. As future work we envision several optimization techniques that may be added to further improve on the memory usage. A possible candidate for this improvement is the association table, which may be aggressively purged to remove links that will not be needed during the course of the query processing on fragments. Moreover, to improve the running time of XFrag, instead of scheduling the 'process', 'inquire' and 'trigger' operation for each fragment, we could schedule these operations across a group of fragments, as not all fragments will result in triggering other fragments downline. However, there is a tradeoff between the scheduling frequency and the memory consumption as now more fragments may be held in memory before they can be triggered and flushed.

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

8. REFERENCES

[1] C. Beeri and Y. Tzaban. SAL: An Algebra for Semistructured Data and XML. In *WebDB'99, Philadelphia, Pennsylvania*, pages 37–42, June 1999.

$$\mathcal{P}[\![\rho^{url}_{s,p,\omega,t}(\{\})]\!]_\delta \rightarrow \{\, r \mid f \leftarrow \mathbf{read}(url), r \leftarrow \mathcal{P}[\![s(f)]\!]_\delta \} \quad (R1)$$

$$\begin{aligned}\mathcal{P}[\![\mu^{path,c}_{s,p,\omega,t}(f)]\!]_\delta \rightarrow\ & \{\, r \mid f/@tsid \neq t, r \leftarrow \mathcal{P}[\![s(f)]\!]_\delta \} \bigcup \mathcal{P}[\![c(f)]\!]_\delta \bigcup \\ & \{\, r \mid fid \leftarrow f/@tsid, fid = t, i \leftarrow \mathcal{I}[\![\mu^{path,c}_{s,p,\omega,t}(f/@id)]\!]_\delta, \\ & \quad \omega \leftarrow \omega \circ (fid, f/hole/@id, (i?\text{true} : (s = \phi?f/path : \perp))), \\ & \quad r \leftarrow (s = \phi?(i?(f/path, fid) : \{\}) : (\mathcal{P}[\![s(f)]\!]_\delta \cup (i?\mathcal{T}[\![s(fid)]\!]_\delta : \{\}))) \}\end{aligned} \quad (R2)$$

$$\begin{aligned}\mathcal{P}[\![\sigma^{lhs,rhs,op}_{s,p,\omega,t}(f)]\!]_\delta \rightarrow\ & \{\, r \mid f_l \leftarrow \mathcal{P}[\![lhs(f)]\!]_\delta, f_r \leftarrow \mathcal{P}[\![rhs(f)]\!]_\delta, op(f_l, f_r), \\ & \quad fid_p \leftarrow \mathcal{M}(f_l, f_r), r \leftarrow \mathcal{T}[\![p(fid_p)]\!]_\delta \}\end{aligned} \quad (R3)$$

$$\mathcal{P}[\![\{\beta_1, \beta_2, \ldots, \beta_n\}(\{f_1, f_2, \ldots, f_m\})]\!]_\delta \rightarrow \bigcup_{\substack{1 \leq j \leq m \\ 1 \leq i \leq n}} \mathcal{P}[\![\beta_i(f_j)]\!]_\delta \quad (R4)$$

$$\mathcal{I}[\![\beta_{s,p,\omega,t}(fid)]\!]_\delta \rightarrow \vee/\{\, e.v \mid e \leftarrow p.\omega, hid \leftarrow e.hids, hid = fid \} \quad (R5)$$

$$\mathcal{I}[\![\mu^{path,c}_{s,p,\omega,t}(fid)]\!]_\delta \rightarrow \mathcal{I}[\![\beta_{s,p,\omega,t}(fid)]\!]_\delta \wedge (c = \phi\,?\,\text{true} : \mathcal{I}[\![c(fid)]\!]_\delta) \quad (R6)$$

$$\mathcal{I}[\![\{\beta_1, \beta_2, \ldots, \beta_n\}(fid)]\!]_\delta \rightarrow \bigvee_{1 \leq i \leq n} \mathcal{I}[\![\beta_i(fid)]\!]_\delta \quad (R7)$$

$$\begin{aligned}\mathcal{T}[\![\beta_{s,p,\omega,t}(fid)]\!]_\delta \rightarrow\ & \{\, (e.fid, e.v) \mid \mathcal{I}[\![\beta(fid)]\!]_\delta, s = \phi, e \leftarrow \omega, fid = e.fid \} \bigcup \\ & \{\, \mathcal{T}[\![s(hid)]\!]_\delta \mid \mathcal{I}[\![\beta(fid)]\!]_\delta, s \neq \phi, e \leftarrow \omega, fid = e.fid, hid \leftarrow e.hids \}\end{aligned} \quad (R8)$$

$$\mathcal{T}[\![\{\beta_1, \beta_2, \ldots, \beta_n\}(fid)]\!]_\delta \rightarrow \bigcup_{1 \leq i \leq n} \mathcal{T}[\![\beta_i(fid)]\!]_\delta \quad (R9)$$

Figure 4: The XFrag Processing Model

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

[3] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *VLDB 2004*, pages 1309–1312.

[4] D. Florescu, et al. The BEA/XQRL Streaming XQuery Processor. In *VLDB 2003*.

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

[6] 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.

[7] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski, "Streaming xpath processing with forward and backward axes." in *ICDE*, 2003, pp. 455–466.

[8] D. Olteanu, T. Furche, and F. Bry. An Efficient Single-Pass Query Evaluator for XML Data Streams. In *SAC 2004*, March 2004, Nicosia, Cyprus.

[9] F. Peng and S. Chawathe. XPath Queries on Streaming Data. In *SIGMOD 2003*, pp 431-442.

[10] Qizx/Open. At `http://www.xfra.net/qizxopen`.

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

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

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

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

[15] L. Fegaras. The Joy of SAX. In *XIME-P 2004*, pages 51–65. June 2004.

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

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

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

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

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

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

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

[23] S. Bose, L. Fegaras. Data Stream Management for Historical XML Data. In *SIGMOD 2004*, pages 239–250, June 2004.

[24] E. Wang, et al. Efficient Management of XML Contents over Wireless Environment by Xstream. In *ACM-SAC 2004*, pages 1122–1127, March 2004.

[25] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou, "A transducer-based xml query processor." in *VLDB*, 2002, pp. 227–238.

[26] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy Evaluation for Active XML. In *SIGMOD 2004*, pages 227–238, June 2004.