EFFICIENT XQUERY PROCESSING OF STREAMED XML FRAGMENTS

by

SEO YOUNG AHN

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE & ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2005

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation and gratitude to Dr. Leonidas Fegaras, Mr. David Levine and Dr. Ramez Elmasri. This thesis work could not have been possible without their constant guidance, patience and motivation.

I want to thank my parents, Sil Soo Ahn and Myung Jin Lee, and my friend Larry DeCoux for all their support and help. Without their support and love, I would not be able to complete my study at the University of Texas at Arlington.

This acknowledgement is incomplete without mentioning my lab members who helped me throughout the work.

November 1, 2005

ABSTRACT

EFFICENT XQUERY PROCESSING OF STREAMED XML FRAGMENTS

Publication No. _____

Seo Young Ahn, M.S.

The University of Texas at Arlington, 2005

Supervising Professor: Leonidas Fegaras

XStreamCast is a push-based streamed XML query processing system that supports multiple servers and clients. The servers broadcast streamed XML data while the clients register to these servers for a specific service and process streamed XML fragments.

This thesis presents methods for efficient XQuery processing of streamed XML fragments for the client. The XQuery parser parses the XQuery given by the

user first. The client processes the fragments and stores only the needed data for the query. The query is then applied to stored XML fragments. This system can be valuable for managing the memory of the client because it does not have to deal with the entire XML document.

This thesis shows the way to handle XQuery and XPath queries. Finally, this thesis shows the experimental results verifying our method for handling XQuery and XPath by comparing it with JAXP, which is the Java API for XML Processing.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

Table Page

CHAPTER I

INTRODUCTION

As the Internet is being expanded, people are becoming more aware of the need for handling both the structure and the content of complex data conveniently and safely. XML is a versatile, simple, and very flexible semi-structured language for representing and exchanging a wide variety of data on the Web and diverse sources, such as structured and semi-structured documents, relational databases, etc. This very extensible text format language is derived from SGML (Standard Generalized Markup Language). It was developed by XML working Groups, supported by the World Wide Consortium (W3C) in 1996. Now it has become a significantly powerful language for storing and transmitting data across diverse application domains.

XQuery [2], an XML Query Language which were invented by the World Wide Web Consortium (W3C), offers an effective and standardized way to query any kind of XML information. There are many systems that support XQuery queries. However, the XStreamCast system is designed for the XQuery processing of

continuously data streams. It processes XML fragments which are transmitted by the server.

XStreamCast is a push-based continuously streamed XML query processing system. It supports multiple servers and clients. As it is push-based, the servers broadcast streamed XML data, which is in the form of XML fragments, to the clients concurrently while the clients tune-in to the streamed XML fragments and evaluate XML queries against that data.

The information of XML data can be attained from a weather report, stock market, relational databases, or text documents. The clients can be networked devices, such as cell phones, PDAs, labtops etc., as long as they are able to connect to the network and provide some storage for XML data.

Once the clients get the XML data transmitted by the servers, they analyze the query that was submitted by the user first. That is, the user query is parsed and converted appropriately to an evaluation query depending on its syntaxes. The XML fragments are analyzed and only the ones useful to the query are stored in the client's memory. The query is, then applied to the stored XML data.

This thesis presents an efficient way for processing XQuery given by the user by using an XQuery processing algorithm, which gets abstract syntax trees as

input and converts them into suitable forms for processing.

CHAPTER II

PROBLEM DEFINITION

An XStreamCast client processes the continuously incoming streamed data which is in the form of XML fragments while these clients receive and analyze incoming data, the server continuously broadcasts the data streams and also periodically broadcasts the structure summary of the original XML document. The structure summary of the incoming stream is called the tag structure and lets clients know about the form of the data stream and is broadcasted in a particular format in agreement with the clients. An XStreamCast client gets a tag structure, streamed XML fragments, and a query given by the user as an input, and it outputs the desired query results in XML.

The streamed XML data can be taken from user interest sources, relational databases, documents of any kind. The clients of XStreamCast can be any networked device and must provide some storage for XML data.

XStreamCast clients provide efficient XQuery processing in continuous XML data and produce the output correctly in an optimizing way to improve query

throughput and response time under the limited resources of clients.

The main problem that we address is about how to support the predicates of XPath and specific forms of XQuery and improve the given query throughput and the response time. This thesis especially crystallizes the processing of the fragments for multiple and nested predicates of XPath and the FLWR expressions of XQuery. Our approach of handling these is explained in chapter 4.

CHAPTER III

THE XSTREAMCAST SERVER

XStreamCast is a push-based, light-weight, in-memory database, continuously operating on streamed XML data. This system contains multiple servers and clients. Since it is push-based, the servers broadcast streamed XML data, which is in the form of XML fragments, to the clients concurrently and the clients receive the streamed XML fragments and process XML queries against that data.

Once the clients get the XML data transmitted by the servers, they analyze the query that was submitted by a user first by using an XQuery processing algorithm. That is, the XQuery, given by the user, is parsed by the XQuery Parser in the XQuery Optimizer component and transformed appropriately for evaluation by the XQuery processing algorithm described in chapter 4. The XML fragments useful to the given query are stored by the client. The query is then applied to the stored XML data.

A XStreamCast server fragments of an XML document, generating the structure summary of the original XML document, called the tag structure and

broadcasting the data fragments and tag structure to multiple clients. The XStreamCast server can be any networked device that can receive the data from a set of data sources, such as sensors that report weather, stock market data, mobile agents collecting network statistics, or certain stored relational databases, with considerable amount of memory to buffer data from a set of data sources over a period of time [3]. Figure 3.1 is the architecture of the XStreamCast server.

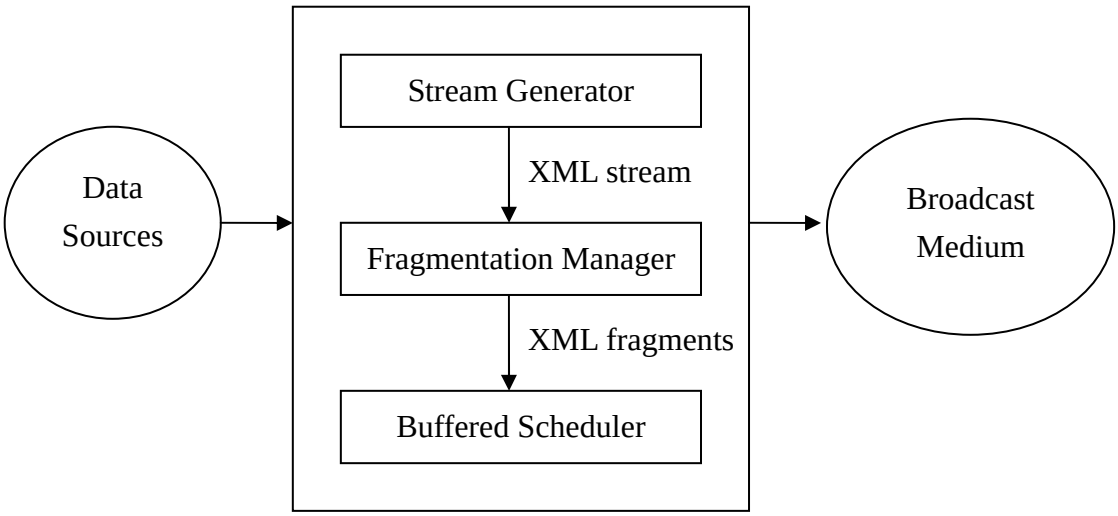

Figure 3.1: Architecture of the XStreamCast Server

As shown in figure 3.1, the XStreamCast server consists of three components, which are the Stream Generator, Fragmentation Manager and Buffered Scheduler. These components are explained briefly. The details are in paper [3].

The stream generator connects and collects the data sources. The XML stream provided by the stream generator is the input and a set of fragments is the output of the fragmentation manager. XML data can be represented by a tree. Each node indicates an element and each directed edge from the parent node to the child node is the parent-child relationship between elements. The tag name of each element is the label of the node and the leaf nodes give the textual data. Figure 3.2 is an XML document and figure 3.3 is the tree representation of the XML data in figure 3.2.

```
<department>
        <deptname>Computer Science</deptname>
        <gradstudent>
                <name>
                        <lastname>Chang</lastname>
                        <firstname>Richard</firstname>
                </name>
                <phone>2626612</phone>
                <email>pdc@cs.wisc.edu</email>
                <address>
                        <city>Madison</city>
                        <state>WI</state>
                        <zip>53706</zip>
                </address>
                <office>5384</office>
                <url>www.cs.wisc.edu/~pdc</url>
                <gpa>3.5</gpa>
        </gradstudent>
        <undergradstudent>
                <name>
                        <lastname>Wagner</lastname>
                        <firstname>James</firstname>
                </name>
                <phone>2626634</phone>
                <email>wagner@cs.wisc.edu</email>
                <address>
                        <city>Madison</city>
                        <state>WI</state>
                        <zip>53705</zip>
                </address>
                <gpa>3.5</gpa>
        </undergradstudent>
</department>
```

Figure 3.2: An XML document

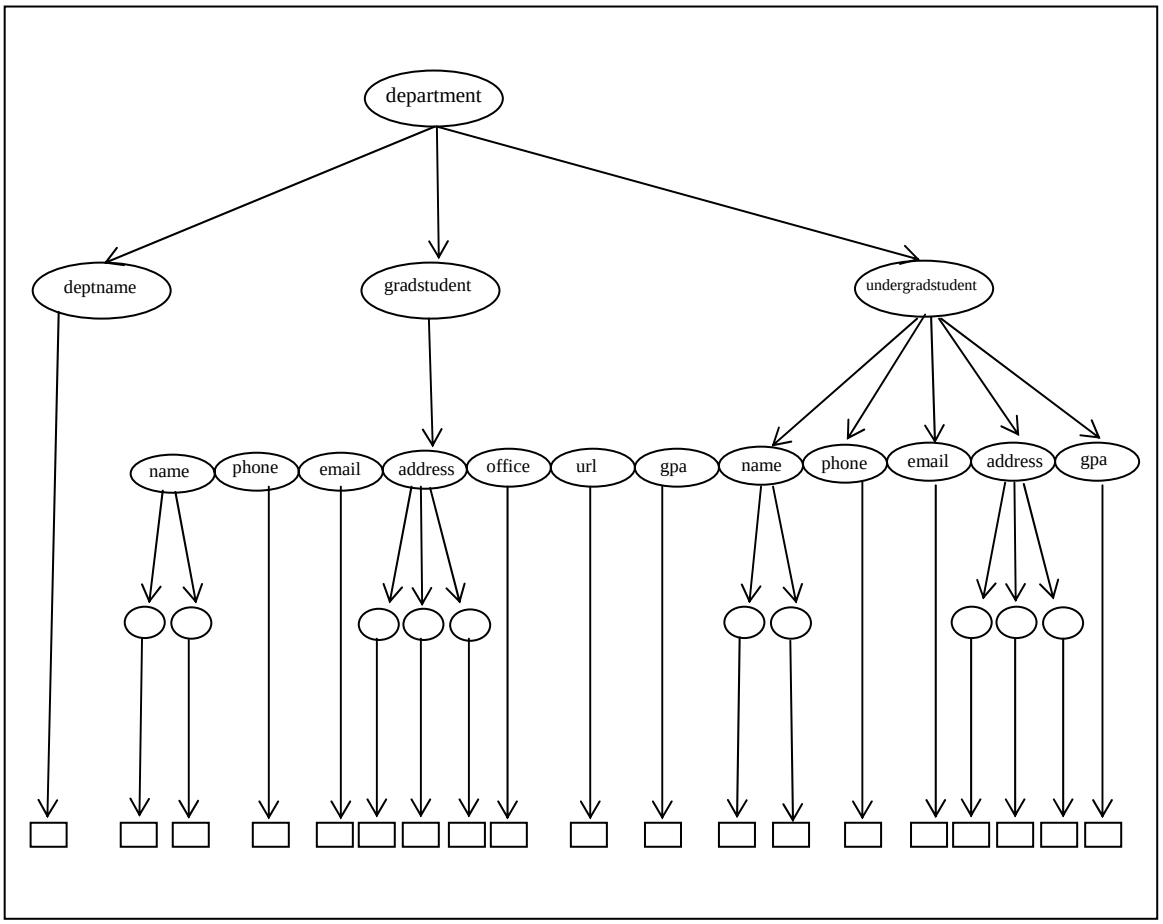

Figure 3.3: Tree Representation of the XML data

The fragmentation of the XML document in figure 3.2 generates a set of subtrees given in figure 3.4 and figure 3.5 that break down into smaller fragments.

<stream:filler id=335 tag=name sid=335> <hole id=336> </hole> </stream:filler>
<stream:filler id=3 tag=name sid=3> <hole id=4> </hole> </stream:filler>
<stream:filler id=246 tag=gradstudent sid=246> <hole id=256> </hole> </stream:filler>
<stream:filler id=46 tag=address sid=46> <hole id=49> </hole> </stream:filler>
<stream:filler id=441 tag=undergradstudent sid=441> <hole id=445> </hole> </stream:filler>
<stream:filler id=233 tag=gradstudent sid=233> <hole id=239> </hole> </stream:filler>
<stream:filler id=118 tag=gradstudent sid=118> <hole id=123> </hole> </stream:filler>
<stream:filler id=306 tag=staff sid=306> <hole id=310> </hole> </stream:filler>
<stream:filler id=233 tag=gradstudent sid=233> <hole id=245> </hole> </stream:filler>
<stream:filler id=2 tag=gradstudent sid=2> <hole id=12> </hole> </stream:filler>
<stream:filler id=66 tag=gradstudent sid=66> <hole id=78> </hole> </stream:filler>
<stream:filler id=247 tag=name sid=247> <hole id=248> </hole> </stream:filler>
<stream:filler id=246 tag=gradstudent sid=246> <hole id=257> </hole> </stream:filler>
<stream:filler id=246 tag=gradstudent sid=246> <hole id=251> </hole> </stream:filler>
<stream:filler id=53 tag=gradstudent sid=53> <hole id=54> </hole> </stream:filler>
<stream:filler id=293 tag=name sid=293> <hole id=294> </hole> </stream:filler>
<stream:filler id=28 tag=gradstudent sid=28> <hole id=32> </hole> </stream:filler>
<stream:filler id=34 tag=address sid=34> <hole id=37> </hole> </stream:filler>
<stream:filler id=79 tag=gradstudent sid=79> <hole id=90> </hole> </stream:filler>
<stream:filler id=92 tag=gradstudent sid=92> <hole id=102> </hole> </stream:filler>
<stream:filler id=272 tag=gradstudent sid=272> <hole id=273> </hole> </stream:filler>
<stream:filler id=79 tag=gradstudent sid=79> <hole id=85> </hole> </stream:filler>
<stream:filler id=420 tag=name sid=420> <hole id=422> </hole> </stream:filler>
<stream:filler id=175 tag=address sid=175> <hole id=177> </hole> </stream:filler>
<stream:filler id=430 tag=undergradstudent sid=430> <hole id=434> </hole> </stream:filler>

Figure 3.4: XML fragments for the University XML document

The fragmentation generator is based on the hole-filler concept [5][6]. The hole is a node of a subtree in the XML tree and represented by the filler. Every hole has a unique hole ID which is a unique reference to a filler. That is, the unique filler fits into the hole. Each filler stands for a rooted subtree in the original XML tree. Therefore, each fragment of the original XML tree has a hole and the corresponding filler for each node in the XML tree like fragments in figure 3.4 above.

```
<stream:filler id=0 childtag=Actors>1</stream:filler>
<stream:filler id=1 childtag=Actor>2|7</stream:filler>
<stream:filler id=2 childtag=Name>3|5</stream:filler>
<stream:filler id=3 childtag=FirstName>4</stream:filler>
<stream:filler id=4 tag=FirstName>Frank</stream:filler>
<stream:filler id=5 childtag=LastName>6</stream:filler>
<stream:filler id=6 tag=LastName>Albertson</stream:filler>
<stream:filler id=7 childtag=Filmography>8</stream:filler>
<stream:filler id=8 childtag=Movie>9|11</stream:filler>
<stream:filler id=9 childtag=Title>10</stream:filler>
<stream:filler id=10 tag=Title>Bye Bye Birdie</stream:filler>
<stream:filler id=11 childtag=Year>12</stream:filler>
<stream:filler id=12 tag=Year>1963</stream:filler>
```

Figure 3.5: XML fragments for the Actors XML document

Figure 3.5 shows the XML fragments for the Actors XML document. 0 through 12 are the unique filler ids and 1 through 12 are the unique hole ids. The filler with id 0 is the root filler that is also the root of the fragments.

XStreamCast processes the continuously streamed XML fragments. Clients get the fragments broadcasted by servers, so they need to know the structure of the original XML document, not to alter the information but to recover it. Thus the servers periodically broadcast along with the data streams and the structure summary of the primary XML document to the clients, called the Tag Structure, which is similar to an XML schema. The tag structure has a tag structure id for each node, called a sid. Each fragment in the data stream has a tag structure sid to help query

processing. Figure 3.6 and 3.7 show the tag structure.

```
<0 name=department>
        <1 name=deptname></1>
        <2 name=gradstudent>
                <3 name=name>
                        <4 name=lastname></4>
                        <5 name=firstname></5>
                </3>
                <6 name=phone></6>
                <7 name=email></7>
                <8 name=address>
                        <9 name=city></9>
                        <10 name=state></10>
                        <11 name=zip></11>
                </8>
                <12 name=office></12>
                <13 name=url></13>
                <14 name=gpa></14>
        </2>
        <15 undergradstudent>
                <16 name>
                        <17 lastname></17>
                        <18 firstname></18>
                </16>
                <19 phone></19>
                <20 email></20>
                <21 address>
                        <22 city></22>
                        <23 state></23>
                        <24 zip></24>
                </21>
                <25 gpa></25>
        </15>
</0>
```

Figure 3.6: Tag structure for the University XML document

```
<0 name=Actors>
        <1 name=Actor>
                <2 name=Name>
                        <3 name=FirstName></3>
                        <4 name=LastName></4>
                </2>
                <5 name=Filmography>
                        <6 name=Movie>
                                <7 name=Title></7>
                                <8 name=Year></8>
                        </6>
                </5>
        </1>
</0>
```

Figure 3.7: Tag structure for XML tree

In figure 3.7, the tag structure ids 0 through 8 are unique. Once the client receives the data from the server, the client retrieves the tag structure first and then analyzes the fragments that contain the tag structure id ‘sid’ and stores or discards the fragments depending upon the given query. This sid is needed by clients to recognize all the valid paths like the path example ‘Actors/Actor/ Filmography/Movie/Titile’ and identify the node of the tag structure each fragment belongs to. Figure 3.8 shows the fragments with the tag structure ids.

```
<stream:filler id=0 childtag=Actors sid=0>1</stream:filler>
<stream:filler id=1 childtag=Actor sid=1>2|7</stream:filler>
<stream:filler id=2 childtag=Name sid=2>3|5</stream:filler>
<stream:filler id=3 childtag=FirstName sid=3>4</stream:filler>
<stream:filler id=4 tag=FirstName sid=3>Frank</stream:filler>
<stream:filler id=5 childtag=LastName sid=4>6</stream:filler>
<stream:filler id=6 tag=LastName sid=4>Albertson</stream:filler>
<stream:filler id=7 childtag=Filmography sid=5>8</stream:filler>
<stream:filler id=8 childtag=Movie sid=6>9|11</stream:filler>
<stream:filler id=9 childtag=Title    sid=7>10</stream:filler>
<stream:filler id=10 tag=Title sid=7>Bye Bye Birdie</stream:filler>
<stream:filler id=11 childtag=Year sid=8>12</stream:filler>
<stream:filler id=12 tag=Year sid=8>1963</stream:filler>
```

Figure 3.8: Fragments with the tag structure ids

CHAPTER IV

THE XSTREAMCAST CLIENT

A client of the XStreamCast system analyzes the query given by the user, processes the XML fragments transmitted by the server and gives the user the results in XML format. The given query is in the form of XQuery. A client can be any networked device, if it has some storage for XML data and is available to connect to the network. The XStreamCast client contains various components, which are the translation engine, query optimizer, and data manager. These components are explained briefly. Their details and their functions are in paper [3].

First the client application obtains the specific query from the user, which is in the form of XQuery. The translation engine component transforms the XQuery query into an algebraic form, and this form is converted into a query plan by the query optimizer component and added into the set of the query plans. These queries in the set of the query plans are scheduled for evaluation [3]. Next the data manager component collects the XML fragments broadcasted by the server through the network and decides whether to store or discard the fragments by the data manager component based on the needs of the query given by the user. Finally the user

obtains the results from the client application.

This thesis makes some assumptions for evaluating the application on the client. There is one single client who gets one stream only, from a single server. Furthermore, this system can execute XPath and XQuery representation. Figure 4.1 shows the architecture of the client.

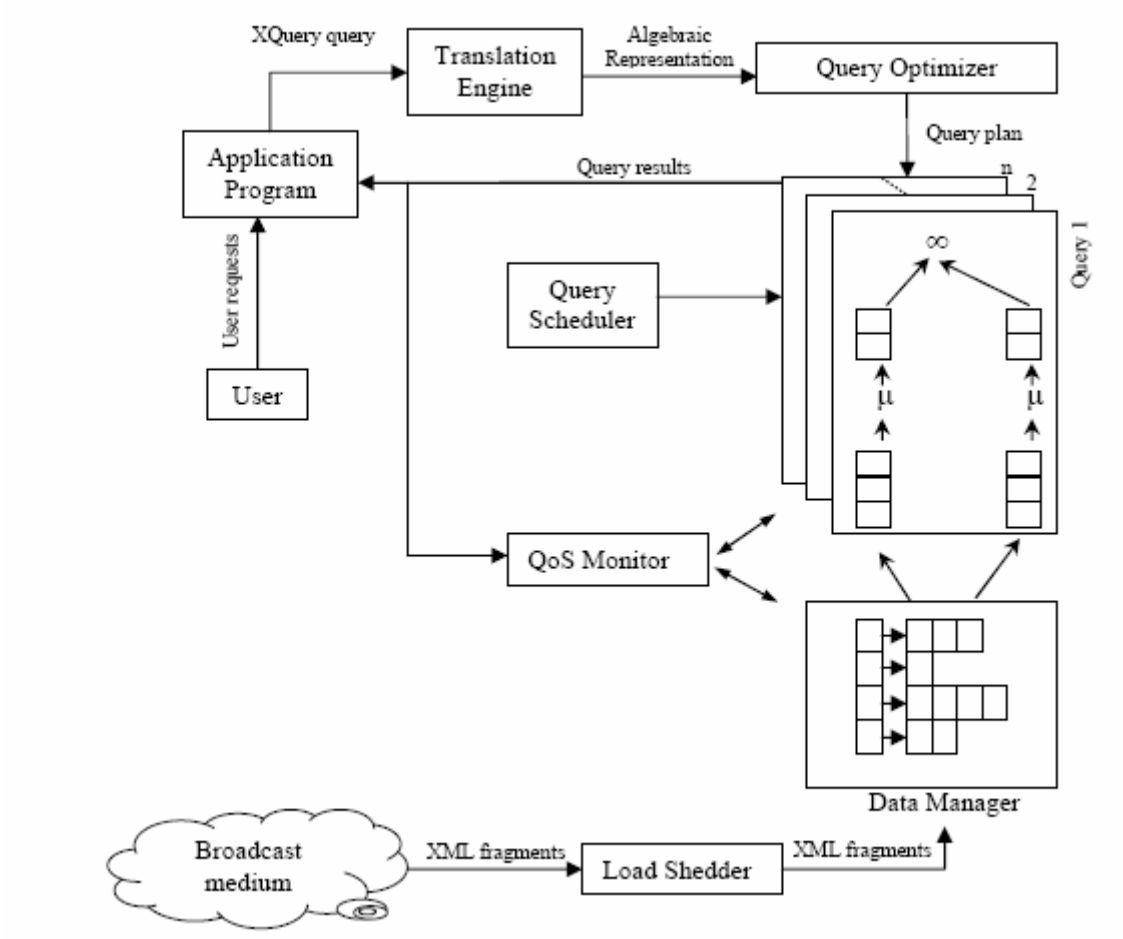

Figure 4.1: Architecture of the XStreamCast Client

4.1 XQuery Parsing and Processing

The XQuery Parser is a part of the Query Optimizer component within the client. This parser uses the Gen package to build abstract syntax trees. The Gen, a Java preprocessor, is one of the Java packages for constructing and manipulating abstract syntax trees. The abstract syntax tree (ASTs) is a data structure that looks like a tree. Examples of ASTs are given in figure 4.2 below. The following examples are the output of the XQuery parsing when the XPath or XQuery query is input by the user. The XQuery Parser obtains the XQuery queries as input and returns the ASTs.

```
Query: //Location
Abstract Syntax Tree: path(descendant(Location))

Query: for $a in document("schema.txt")//Text
        where $a/Keyword=” express"
        return <Q> { $a } </Q>
Abstract Syntax Tree:
clauses(for(a,path(child(call(document,"schema.txt")),descendant(Text))),
            where(call(eq,path(child(variable(a)),child(Keyword)),"express")),
            orderby(),
            element(Q,concatenate(concatenate(" ",path(child(variable(a)))),” )))
```

Figure 4.2: Abstract Syntax Tree

The current version of the parser accepts *for* and *let* clauses, single and multiple clauses, path expressions and multiple nested predicates only.

query1 : path expressions
Query: //Location

Query1 above is a step in XPath, which is called the descendent. A single slash ‘/’ means the step of each path, and is called a child. Like the query1, if there is a double slash ‘//’, it expresses the relation under the tag name followed by ‘//’. This path expression selects all descendent elements with tag name ‘Location’.

query2 : predicates
Query: //Text[Keyword = "express"]/Emph

Query2 represents the predicate which is a condition of the following query above. The query returns results only if the condition is true, so it limits the extracted data from XML data. This predicate is used to select all the Emph elements under the Text element whose Keyword element is equal to the text ‘express’. That means this query returns all the children of the node ‘Emph’, if the predicate is true.

query3 : nested predicates
Query: //Mail[Text[Keyword = "express"]/Emph = "overnight delivery"]/Text

If there is another predicate within a predicate, called a nested predicate, the

condition has to be true for both in order to get a non-empty result. This query retrieves all the Text elements under the Mail element, and both /Text/Keyword which is equal to the text ‘express’ and /Text/Emph which is equal to the text ‘overnight delivery’ are true.

The examples query1 through 3 are XPath queries which are a subset of XQuery. Next, I will present the XQuery FLWR expressions, which are pronounced as ‘flower expressions’. These FLWR expressions are one of the powerful expression types of the XQuery, because they give the users convenience for using the XQueries in a SQL-like format. It also makes the use of other kinds of expressions, such as constructions, conditional and logical expressions etc. even easier. When adding an ORDER BY expression, which defines the sort order, the name becomes FLOWR. However, currently this thesis does not handle an ORDER BY clauses. I used the document ‘schema.txt’ which contains a tag structure and XML fragments. The ‘$’ represents a variable which usually gets a value from an expression in the first few lines of the query.

query4 : FLWR Expression for the for clause
Query: for $a in document("schema.txt")//Text
where $a/Keyword=” express"
return <Q> { $a } </Q>

Query4 uses the *for, where,* and *return* clauses. The *for* clause binds a variable to each item returned by the expression and results in iteration. In the query4, the *for* selects all Text elements within the indicating document ‘schema.txt’ and binds it to a variable $a. The *where* clause signifies a condition expressed in XPath. In the query above, the *where* clause selects all Text elements, if the Keyword element under the variable $a is equal to the text ‘express’. The *return* clause specifies the format to be returned, which means a sequence of the results. Adding <Q> and </Q> tags to the FLWR expressions lists all the results in the element with tag. Query4 returns all $a whose condition is true inside the element with tag <Q>.

query5 : FLWR Expression for the let clause
Query: let $a := document("schema.txt")//Country
return $a/Item

The *let* clause signifies that a new variable has a specified value. That is, the *let* clause allows variable assignments, but does not iterate over sequence values, unlike the *for* clause. In the abstract syntax tree above, the variable $a is assigned to be all the descendents of the element Country. The *where* and *orderby* clauses are empty and the return path is the child of Item that is stored under the variable $a.

query6 : FLWR Expression for the for clause with the returning tags
Query: for $a in document("schema.txt")//Country
return <Q> { $a/Item } </Q>

Query6 is a *for* clause that returns the same result as query5, but the result now is expressed between <Q> and </Q> tags in iteration.

query7 : FLWR Expression for the for clause with the nested predicate
Query:
for $a in document("schema.txt")//Mail[Text[Keyword = "express"]/Emph = "overnight delivery"]
return <Q> { $a/Date} </Q>

Query7 is an example of having both a for clause and a nested predicate. The variable $a is the path under the element Mail satisfying both conditions.

query8 : FLWR Expression for the for clause with the nested predicate within the returning tag
Query:for $a in document("schema.txt")//Item
return <Q>
{ $a//Mail[Text[Keyword = "express"]/Emph = "overnight delivery"]/Date}
</Q>

The result of query8 is similar to that of query7. In this case, the specific nested conditions are inside the return clause.

This thesis presents a method for processing XQuery and XPath queries, which is the subset of the XQuery. The Input is the Abstract Syntax Tree(AST)

generated by the XQuery Parser. Figure 4.3 represents the XQuery processing algorithm.

if the next token is *clauses* and the id is *for* or *let* clause
1. store the variable and the id
2. find the first token which is *child* or *descendant* in last of Asts
    a. store the tagname of *child* or *descendant* and the path
3. if the next token is *where*
    a. store the operator which is eq, neq, lt, gt etc.
4. if the next token is *element*
    a. store the return tag into the *returntag* and the conditions
5. if the next token is *path*
    a. find the *variable*
    b. compare the variable to the variable stored at step1 and store the path
  else if the next token is *clauses*
    a. find the *variable*
    b. compare the variable to the stored variable and store the path
    c. if the variable is different from the old variable stored at step1
      then store the new variable and go to the step2
if next token is *path*
if the next token is *child*
6. if the token is a character
    a. if there is no predicate, then evaluate the method *XPathTagged*
    b. else evaluate the method *PredXPathTagged*
7. if the next token is *any* which is '*', a wild card
    a. if there is no predicate, then evaluate the method *XPathAny*
    b. else then evaluate the method *PreXPathAny*
if the next token is *descendant*
8. if the next token is a character
    a. if there is no predicate, then evaluate the method *XPathDescendant*
    b. else then evaluate the method *PredXPathDescendant*
9. if the next token is *any* which is '*', a wild card
    a. if there is no predicate, then evaluate the method *XPathDescendant*
    b. else then evaluate the method *PreXPathDescendant*
    c. if neither, then the *error*
10. if the next token is *condition*
    a. if there is not predicate
        - if it is a character, then evaluate the method *PredXPathTagged*
        - else if it is *any* which is '*', a wild card,
          then evaluate the method *PredXPathAny*
        - if neither, then the error
    b. else
        - if it is a character, then evaluate the method *PredXPathTagged*
        - else if it is *any* which is '*', a wild card,
            then evaluate the method *PredXPathAny*
        - if neither, then the *error*
11. print the result in XML format

Figure 4.3: XQuery Processing Algorithm

The Abstract Syntax Tree, which is produced by the XQuery Parser, always starts in the expression *clauses* if it is an XQuery query. On the other hand, XPath query starts in the expression *path*. In this system, the clauses map into *for* or *let* clause. If it is the *let* clause, it executes only once.

4.2 Data Manager

One of the most important components within the client system is the Data Manager component. This component collects the XML fragments and tag structure transmitted by the server and processes them. Then, the query which is parsed by the XQuery Optimizer component is analyzed, the client then finally provides the results to the user in XML format. In processing the fragments, the DataManager decides which fragments are to be stored or discarded base on the needs of the query. It cannot do any progress until it gets the tag structure for recognizing the structure of the primary XML document.

As mentioned earlier, the DataManager component processes the XML fragments, but it has to get the tag structure first for manipulating the fragments. Once the tag structure is transmitted by the server, then the DataManager starts to decide if the fragments are useful or not for the user query.

The DataManager analyzes the incoming tag structure which lets the client

know about the formation of the data stream and location of the specific element before processing the XML fragments, because it needs to know the structure of the original XML document. Each node has a unique tag structure id, called a sid, so this sid assists the DataManager in recognizing the location of the fragment in the incoming streams. Since the sid is unique, the DataManager can find the correct path against the user given query, even though there is more than one of the same tag names in the incoming streams. The details of the way to retrieve the sids required for processing the fragments are in paper [4]. We have looked over the queries with predicates only whose description was in paper [4] and which I have used in the implementation of this thesis.

The DataManager component processes the given query path step which has been broken down into the path steps by the XQuery processing algorithm. When the fragment is processed, it is stored in memory or discarded based on whether it is useful or not against the query.

In paper [4], some entries were defined for processing the queries with predicates. They are the L.C.P (Lowest Common Parent) node, the intermediate nodes and the leaf nodes. The L.C.P node is usually the first node before the first existing predicate in the query, so it is a common parent for all the predicates in the

query and only one for each query with predicates. The leaf nodes are the nodes not having any child and the range of the intermediate nodes are from lower levels of the L.C.P node to higher levels of the leaf nodes. Figure 4.4 represents the tree that shows the following query with nested predicates.

Query:
/department/gradstudent[Name[FirstName = "Richard"]/LastName = Chang"]/Phone/Home

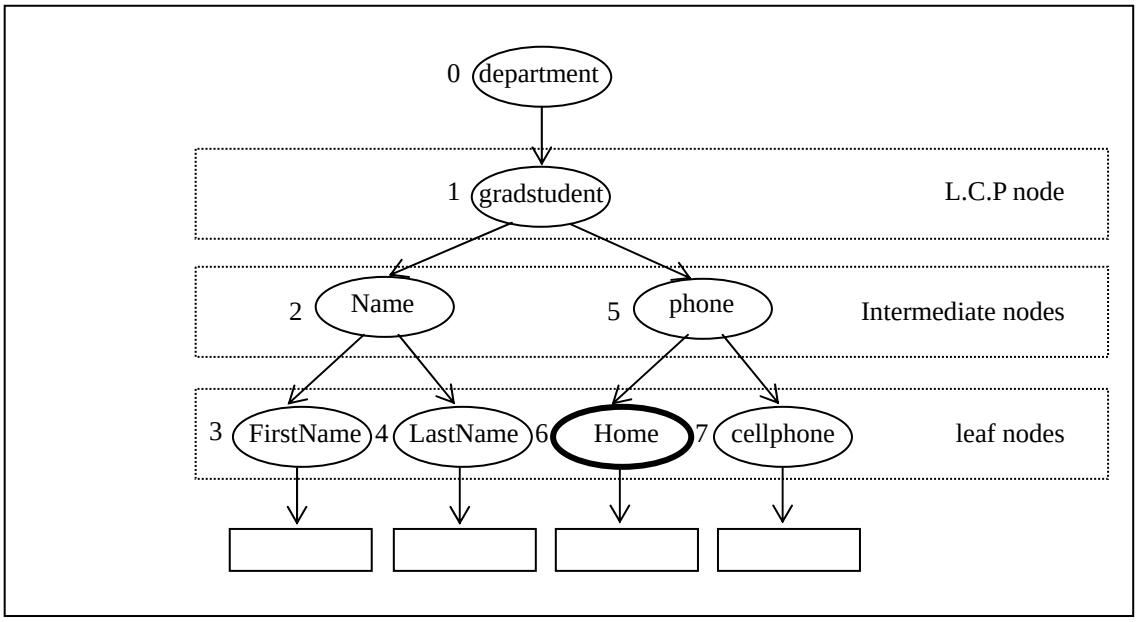

Figure 4.4: Query tree with the nested predicates

In figure 4.4, the tag structure id 1 is the L.C.P node, 2 and 5 are the intermediate nodes and 3 through 7 are the leaf nodes. The element Home with the bold circle is the result. The query above returns the result under the element Home,

if the predicates, both FirstName and Last Name, are satisfied. All elements with tag name Home are then stored. There may be a case where the fragment of the value for the FirstName or LastName element has not arrived yet, so the client can not judge whether to store or not. This is an example of why the client needs the L.C.P node. If the fragment belongs to sid 1, as well as the intermediate fragments that belong to the same L.C.P node should be stored. After all, the DataManager component knows about only the sids which are useful for processing the fragments against the query given by the user.

The algorithm for processing the fragments is in paper [4], so I will only explain the handling of the predicates which I have done in the implementation at this thesis and some data structures for the execution of the predicates.

. As the DataManager has already processed the tag structure ids for the query result, all the useful sids for processing the fragments are stored in the *lcpSids* data structure for the L.C.P id, the *interSids* data structure for the intermediate ids and the *leafSids* data structure for the leaf ids.

The first step is to compare the sid of the incoming fragments to the stored sids. The filler id and hole ids are stored in the *idTable* data structure, representing the relation of the original source, in the form of a hash table, only if the sid of the

fragments is one of the stored sids which are from the tag structure. The *tagTable* data structure in the form of a hash table stores the tag names of the fragments and the *dataTable* data structure in the form of a hash table stores the data fragments. The *parentIds* data structure stores the filler ids, if the sid is same as the one of the ids in the *parentsSids*. Plus, all the fragments which belong to the L.C.P, the intermediate and leaf nodes are stored in *lcpBucket*, *interBucket* and *leafBucket* data structure.

When the DataManager has all the necessary fragments for the predicates, their sids are in the *lcpBucket* if the sid of the fragment is one of the *lcpSids*, in the *interBucket* if the sid of the fragment is one of the *interSids*, and in the *leafBuckets* if the sid of the fragment is one of the *leafSids*. If the L.C.P element in the *lcpBucket* has the predicates satisfied under L.C.P element, then the result of the following fragment of the L.C.P element is the output. To verify that the predicates are satisfied is to test that the value of their child or descendant of that fragment in the *leafBucket* is equal to the value of the predicate. When the query is nested, the DataManager adds all the children and grandchildren from the *interBucket* to the fragment in the *lcpBucket* for each fragment in the *lcpBucket* as its children. The DataManager then checks if it has the same number of children in the *leafBucket* as

the count in the *countIds* for each fragment in the *lcpBucket*. If there are fewer children than the sum, then they are deleted from the query result node from the *parentIds*. The data structure countIds is the counter for all the L.C.P and C.P fragments whose sids are in the countSids. This counter represents the number of the predicates that are satisfied by the L.C.P or C.P node fragments at this moment.

CHAPTER V

RESULTS AND CONCLUSION

5.1 Experimental Setup

XStreamCast provides efficient XQuery processing of streamed XML fragments. It inputs a user query and outputs the results in the form of XML. XStreamCast currently supports one single client who gets one stream only, from a single server.

In this chapter, I evaluate the efficiency of the client of this system against the Java API for XML Processing (JAXP) that provides functionality for reading, manipulating, and generating XML documents through Java APIs. The factors which are considered for the experiments are first response time and memory usage. The first response time per query is defined as the time which is taken for processing a given XPath query by the client. The maximum memory usage is the amount of memory to store only the necessary fragments at the client, especially in the case of the predicates. The input file has a size of 5KB to 5MB and test cases are general XPath queries with child, descendent and predicate steps. These experiments are

executed on a machine with Intel Pentium IV 2.8 GHz processor with 512 MB RAM.

5.2 Test Data Results

In the first experiment, the first response time is the time that is taken for processing XPath queries by the client and is calculated when the client receives all the fragments because the client does not start processing unless it has all the fragments in the current system. The input files are ‘Actors.xml’ and ‘University.xml’. As shown the table 5.1, the first queries are the queries with a child and descendent steps. When the size of the input file is small, such as 5KB and 0.1MB, the first response time is almost the same. The larger the file size, the more the difference in the response time. When the input is more than 1.5MB, the response time difference is almost double and even more as shown in table 5.1.

Table 5.1 First response of the query with descendent

Query	Input File	JAXP	XStreamCast
//Actors	5KB	100	130
//departments	0.1MB	305	292
//departments	1.5MB	911	445
//departments	5MB	1985	783

The figure 5.1 shows the graph of table 5.1. When the input size of the file is less than 0.1MB, the first response time is not different between the XStreamCast and the JAXP.

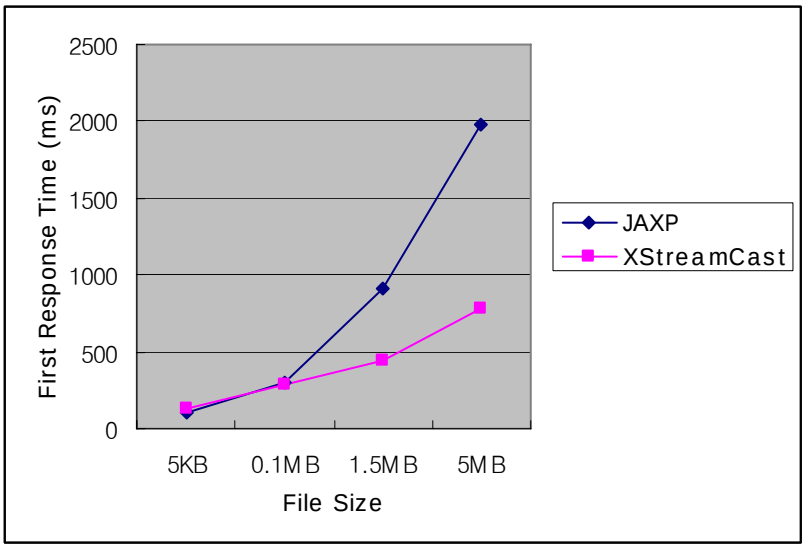

Figure 5.1: First response of the query with descendent.

The second set of examples shows XPath queries with the predicates over the same files as of the first examples. When the size of input file is small such as 5KB, the first response time of the JAXP is a little faster than the first response time of the XStreamCast. As the file size increases, the XStreamCast is getting faster than the JAXP. The graph 5.2 represents table 5.2.

Table 5.2 First response of the query with predicates

Query	Input File	JAXP	XStreamCast
//Actor[Name/LastName="james"]	5KB	120	152
//department[deptname="Linguistics"]	0.1MB	303	206
//department[deptname="Linguistics"]	1.5MB	936	443
//department[deptname="Linguistics"]	5MB	2006	950

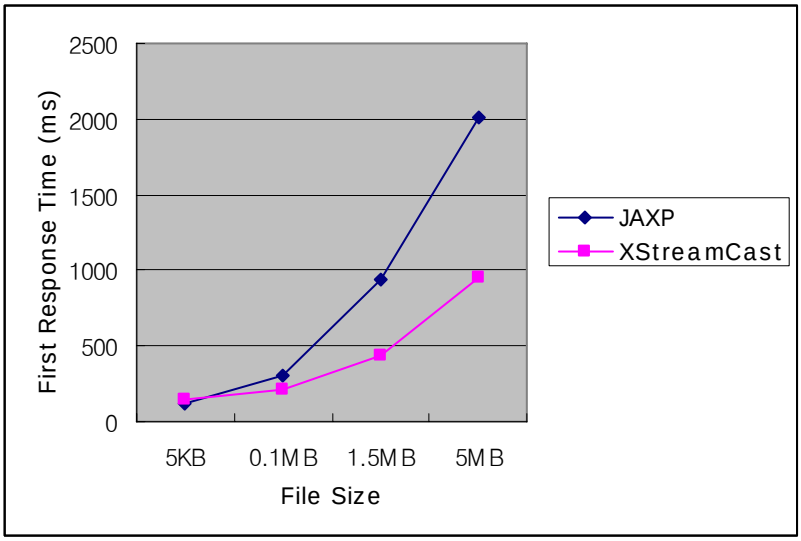

Figure 5.2: First response of the query with predicates

Table 5.3 below shows the first response time with commercial XML data from the XML Data Repository. TPC-H Relational Database Benchmark from Transaction Processing Performance Council takes less query processing time compared to the input files of 5MB in table 5.1 and 5.2. The maximum depth of the input file 'University.xml' is 5 and the maximum depth of the TPC is 3. When the depth is deeper, the server generates more fragments to connect each other and the

client needs more time to process those fragments. Consequently, the depth of the XML data causes the different result even though they have similar file sizes.

Table 5.3 First response of the query with commercial XML data

Input	JAXP	XStreamCast
SIGMOD Record (468 KB)	801	357
TPC-H Relational Database Benchmark (5.2MB)	999	650

In the second experiment, the maximum memory usage at the client is the space consumed to store the fragments and data structure when the query is with the predicates. The graph in figure 5.3 shows the comparison between the XStreamCast and the JAXP. It represents the XStreamCast requires less memory with the query size. The JAXP requires more memory as they parse and process the XML document. As increasing the file size, the difference in memory usage is getting almost the same.

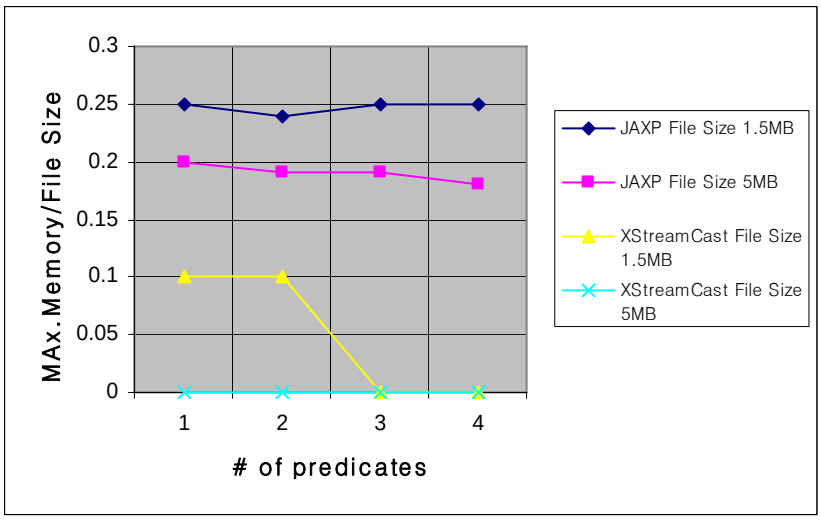

Figure 5.3: Max. Memory usage/File size vs. # of predicates

5.3 Conclusion

As shown in the tables and graphs above, the XStreamCast is much faster than the JAXP except for the small size of the input file because the JAXP processes the query after the whole XML document is parsed. It also reduces the memory requirement as storing only the useful fragments for the query.

CHAPTER VI

RELATED AND FUTURE WORK

6.1 Related work

XML stream processing is an emerging application, so there are a number of recent works for query processing and stream management.

The STREAM Project [9] by the database group at Stanford University is a general-purpose prototype that investigates data management and query processing over continuous unbounded streams of data. This system supports a large class of declarative continuous queries over continuous streams and traditional stored data sets. They developed the language ‘CQL’, a concrete declarative query language for continuous queries over streams and relations.

The Aurora System [10] is an experimental data stream management system that contains a graphical development environment and a runtime system. It has been designed for monitoring applications that deals with very large numbers of continuous data streams from such sources as sensors, satellites and stock feeds. This system provides a user interface for tapping into pre-existing inputs and

network flows and for wiring boxes together to generate the result as outputs.

SPEX [11] evaluates XPath queries against XML data streams. This system processes four steps. First, the input XPath query is rewritten into an XPath query without reverse axis, and the forward XPath query is compiled into a logical query plan abstracting out details of the concrete XPath syntax. Then, a physical query plan is generated by extending the logical query plan with operators for determination and collection of answers. In the last step, the XML stream is processed continuously with the physical query plan, and the output stream conveying the answers to the original query is generated progressively. It also provides a practically useful application for monitoring its runtime processes on UNIX, called SPEX viewer.

XSQ [12] is a system for querying streamed XML data using XPath 1.0. This system is designed based on a hierarchical arrangement of pushdown transducers with buffers and it supports all XPath expressions including multiple predicates, closures, and aggregation. Their issue was how to expect the result before the system gets the data required to evaluate the predicates to decide its state. Plus predicates may access different portions of the data and it may contain a recursive structure. So XSQ system buffers the potential result. To design an automaton for evaluating

XPath queries systematically, they use a hierarchical pushdown transducer which is composed of several basic pushdown transducers. The key idea is to use the position of the basic pushdown transducer in the hierarchical pushdown transducer to encode the results of all predicates. Therefore, the buffer operations in the basic pushdown transducers can be determined accordingly. All XML data is also maintained as a token format of stream. So, this engine can decrease memory usages more than others.

The BEA streaming XQuery processor [13] represents a new commercial realization for querying XML streams using an XQuery queries. With this system it is possible to implement the entire XQuery language specification, types and all. This engine is especially for message processing over the streaming XML data. So, XQuery expressions are an efficient internal representation of XML data using streaming execution to the extent possible and the efficient implementation of XQuery transformations that involve the use of many node constructors. The BEA engine which is for evaluating XQuery over XML stream is started with submitting XQuery queries through Java applications and it consumes query results through an XDBC interface which comes from JDBC. Then the query compiler parses and optimizes the query and generates a query plan as a form of a tree of operators. The

runtime system which contains the function and operator library and XML parser and schema-validator interprets the plan. So, the parsed and schema-validated XML message which is incoming XML data executes once and is used in many different XQuery queries without making an additional cost for parsing and schema-validating. Finally, they are made as free variables to queries and bound through the XDBC interface. All XML data is also maintained as a token format of stream, so this engine can make decreased memory usages.

As shown above, there are many systems for stream management and query processing. XStreamCast is different in that it deals with fragments which are a unit for manipulating the XML stream and the standard XQuery language instead of making another query language. The concept of the use of the fragment can have more opportunities in that it can be extended in handling the data. Plus, this system is easier to be accepted since it uses the standard XQuery language

6.2 Future Work

This thesis presents a method for processing XPath queries, which are path expressions and nested predicates, and XQuery queries, which are *for* or *let* clauses with nested predicates and return tags. The query processor needs to be extended to

handle more complex queries, such as nested *for* clauses and sort order expressions. Furthermore, the current project only accepts a single input stream from a single server to a single client. It can be extended to handle multiple streams from multiple servers to multiple clients. The Query Optimizer component containing the XQuery parser is part of the current project but there will be more components to be added like the Query Scheduler, which schedules the set of the query plans generated from the Query Optimizer component, and a QoS monitor which controls the load shedding component that can handle the fragment arrival rate from the servers.

REFERENCES

[1] World Wide Web Consortium. Extensible Markup Language (XML) 1.0. http://www.w3.org/XML. February 2004.

[2] World Wide Web Consortium. An XML Query Language (XQuery) 1.0. http://www.w3.org/TR/2005/CR-xquery-20051103/. November 2005.

[3] Vamsi Krishna Chaluvadi, “Efficient Broadcast of XML Streams in a push bashed Envirnment”. Department of Computer Science and Engineering, University of Texas at Arlington, USA. December 2003.

[4] Darsan Tatuneni, “Efficient Processing of Streamed XML Fragments”. Department of Computer Science and Engineering, University of Texas at Arlington, USA. December 2004.

[5] Sujoe Bose, Leonidas Fegaras, David Levine, Vamsi Chaluvadi. “A Query Algebra for Fragmented XML Stream Data”. 9th International Workshop on Data Base Programming Languages (DBPL), Potsdam, Germany, September 2003.

[6] Leonidas Fegaras, David Levine, Sujoe Bose, Vamsi Chaluvadi. “Query Processing of Streamed XML Data”. 11th International Conference on Information and Knowledge Management (CIKM), November 2002.

[7] Yanlei Diao and Michael J. Franklin. “High-Performance XML Filtering: An Overview of YFilter”. IEEE Data Engineering Bulletin, March 2003.

[8] World Wide Web Consortium. XML Path Language (XPath) 2.0. http://www.w3.org/TR/2005/CR-xpath20-20051103/, November 2005.

[9] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslwwicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. “STREAM: The Stanford Data Stream Management System”. Department of Computer Science, Stanford University, March 2004

[10] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, J. Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, S. Zdonik. “Aurora: A Data Stream Management System”. Brandeis University, Brown University, M.I.T.

[11] Franc¸ois Bry, Fatih Coskun, Serap Durmaz, Tim Furche, Dan Olteanu, Markus Spannagel. “The XML Stream Query Processor SPEX”. Institute for Informatics, University of Munich, Germany, 2005

[12] Feng Peng and Sudarshan S. Chswthe. “XPath Queries on Streaming Data”. SIGMOD 2003

[13] Daniela Florescu and Chris Hillery. “The BEA/XQRL Streaming XQuery Processor”. VLDB 2003

BIOGRAPHICAL INFORMATION

Seo Young Ahn received her Bachelor of Science degree in Information Science at Sangmyung University, Korea 2000. She then pursued Masters degree at The University of Texas at Arlington.