WPI-CS-TR-08-15 December 2008

FSA Cost Comparison for XML Stream Processing

by

Joseph Lapointe and Murali Mani

Computer Science Technical Report Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

FSA Cost Comparison for XML Stream Processing*

Joseph Lapointe, Murali Mani
Worcester Polytechnic Institute
jolapoin@wpi.edu, mmani@cs.wpi.edu

Abstract

As applications stream XML data, a challenge arises to effectively process queries on those streams. Current research focuses on building an efficient automaton to process the XML query or split the query into sub-patterns and perform a structural join to obtain the results. In our research, we consider moving some of the structural joins inside the automaton itself by making the automaton more complex. We use readily available tools, JFlap and Oracle, to measure the costs of these actions. We present the concise procedures and mechanisms used to gather the data and compute the results of our experiments.

Definitions

DDL	Data Definition Language
DFA	Deterministic Finite Automaton
DML	Data Manipulation Language
FSA	Finite State Automata
NFA	Nondeterministic Finite Automaton
PDA	Pushdown Automaton
SQL	Structured Query Language
XML	eXtended Markup Language

* This work is partially supported by the National Science Foundation under Grant No. NSF IIS-0414567

1. Introduction

In today's world there are many important applications that stream real time data in XML format. Such applications include stock market updates and real time news feeds. Searching real time data feeds creates an issue with finding data within those feeds. The applications parsing the data cannot seek forward in the data and cannot return to data already received unless explicitly buffered.

Research in both the industry and academia is focusing on finding efficient solutions for this problem. There are currently two models used by researchers to deal with this problem. The first such model uses finite state automata, while the other model uses tuple-based query algebra.

Existing research on models that use finite state automata produced such systems as the XML Toolkit [3] and the XPush Machine [4]. Both of these systems convert a path expression to an NFA prior to translating it to a DFA. The states and transitions for the DFA are loaded at runtime reducing the size of the computation in memory (this technique known as 'lazy DFA'). Other research, such as the XSQ [1] and XSM [5] systems, use the transducer model to build their automaton. The transducer model operates as a PDA; it reads an input symbol and transitions to another state based on both the input and the symbols in the stack.

Moreover, other research focuses on models that use query algebra, such as YFilter [6], TwigM [8], and [Agrawal and Li]. Databases use tuple-based query algebra to optimize queries. Database schema and XML share similar semantics, thus this approach can apply to XML as well. The Raindrop [10] approach uses both query algebra and finite state automaton. Its approach investigates performing different portions of pattern retrieval inside and outside the automata. Take, for example, query 1.

Query 1: //PLAY/ACT[PROLOGUE]/TITLE

Raindrop splits query 1 into two sub-patterns (//Play/Act/Prologue and //Play/Act/Title). For each sub-pattern, Raindrop has two options; retrieve the pattern inside the automaton or retrieve the pattern outside the automaton. A structural join between the results of the two pattern retrievals ensures the results of the join only contain Titles who's Act contain a Prologue.

In our research, we consider moving some of the structural joins inside the automaton itself by making the automaton more complex. In our example, we have the option of retrieving the pattern's results either from inside or outside the automaton. Then, we perform the structural join to obtain the final results of the query. For query 1, four options are considered; (1) both outside the automaton, (2) sub-pattern 1 inside, sub-pattern 2 outside, (3) vice-versa, and (4) both inside the automaton.

Subsequently, there are tradeoffs to executing these patterns inside and outside the automaton. Moving the structural join inside the automaton increases the size of the automaton. Performing the structural join outside the automaton increases the post processing costs. We are taking the first steps in exploring the significance of these trade-offs.

Roadmap. Section 2 describes our approach for measuring the post processing cost and computing the size of the FSA. Section 3 explains the setup and tools used for the experiments. The results of the experiments are in section 4. Section 5 contains our analysis and conclusion. Finally, the appendix contains the additional tools or programs used throughout the experiments.

2. DFA Cost Comparison

Our research focuses on the paradigm of moving the query processing inside or outside of the automaton. This paradigm incurs an automaton cost when moving the query processing inside the automaton. We only consider the size of the automaton as the automaton cost. Moving the query processing outside the automaton incurs an increase in the post processing cost. We will measure both of these costs. We look at several XML queries and separate each query into its sub patterns, as the Raindrop [10] approach would. Each sub pattern matches a portion of the XML data stream. The comparison of the cost will determine which option is best suited for a given pattern.

For example, the following two queries have three sub patterns each:

```
Query 2:  //PLAY/ACT[TITLE]/SCENE[TITLE]/STAGEDIR
Query 3:  //PERSONAE[TITLE]/PGROUP[PERSONA]/GRPDESCR
```

We use readily available tools to measure the size of the automaton and the post processing costs. We use the JFlap [14] tool to convert the XML query (after some preparation) to a DFA. JFlap allows us to save the nodes and the transitions. Using this data, we calculate the total number of nodes and transitions.

Moreover, we use Oracle to perform the structural join operations, and we use its timing[1] mechanism to measure the post processing cost. Each query may have several sub patterns and the structural join operations may occur in a different order. We will measure the post processing cost for each sub-pattern.

3. Experimental setup

The experiments required for the cost comparison of DFAs require several tools and many steps. This section describes the systems, the tools, and the steps involved for retrieving the results. First, we describe the systems used in our setup. Then, we present the tools that we use to measure the costs. Finally, we explain the procedures we followed to get our experimental results.

3.1 Systems

The systems used are a Windows XP system, the WPI Linux server cccwork2.wpi.edu and the WPI Oracle database. The Windows XP system takes the original Shakespeare XML dataset [15] and transforms it into an intermediate XML file. We use Perl scripts on the WPI Linux server to convert the intermediate data into SQL DDL statements. Using Oracle, we load the data using the SQL DDL statements and execute SQL DML statements to simulate the structural join operations.

[1] In Oracle, 'set timing on'

3.2 Tools

The tools that operate on the Shakespeare XML dataset are the Eclipse SDK and the Saxon XML utilities. The Shakespeare XML dataset does not contain any attributes that could help in joining two XML queries. Thus, we created a Java program, utilizing the Eclipse SDK and the SAX XML reader class, to add a Dewey Ordered 'id' attribute to each element, which solves this problem. Saxon interprets the XML (sub-patterns) queries and produces the results.

We also use the JFlap tool and Oracle. We use JFlap to convert the regular expression representations of the XML query into DFAs. We can then count the number of states and transitions of the original and subset XML queries. We use Oracle to measure the structural join costs for sub-patterns retrieved outside the automaton.

3.3 Procedures

The following subsections describe the procedures, or steps, that get us to a state where we can perform the experiments. At this point in time, we introduce our complex XML query. Query four combines queries two and three to form a larger XML tree displayed in Figure 1.

```
Query 4:  //PLAY[ACT[TITLE]/SCENE[TITLE]/STAGEDIR]/  (cont…)
          (cont …) PERSONAE[TITLE]/PGROUP[PERSONA]/GRPDESCR
```

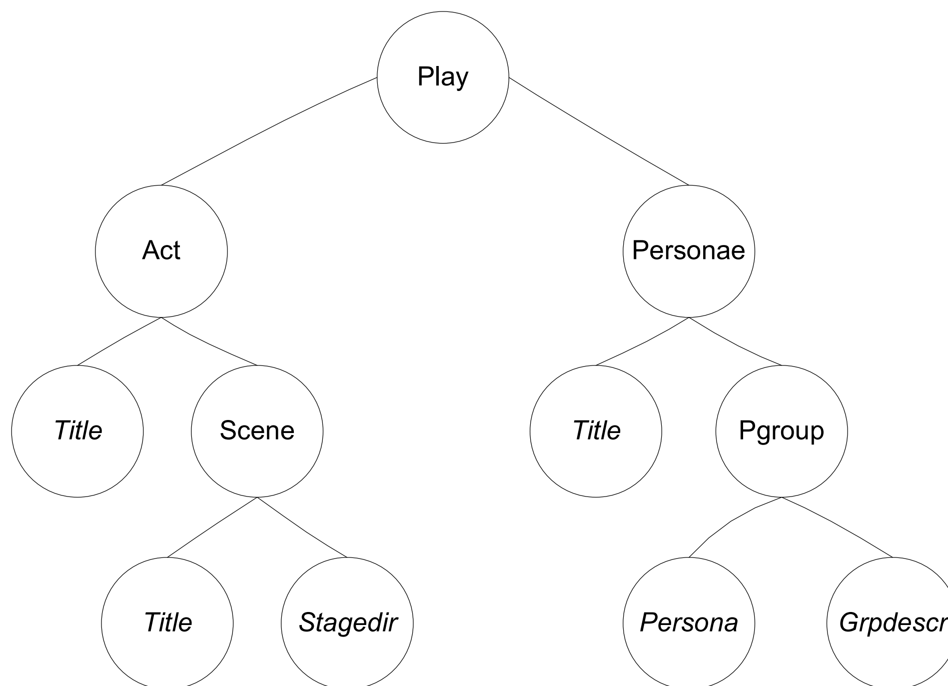

Figure 1. XML Tree for Query 4.

3.3.1 Splitting the XML Query

The XML queries are broken down into sub-patterns. Each path from the start node to a leaf node is a sub-query. Every time an enclosure '[' and ']' is encountered, the tree is branched. In query four, there are six sub-queries, which are:

- //Play/Act/Title
- //Play/Act/Scene/Title
- //Play/Act/Scene/Stagedir
- //Play/Personae/Title
- //Play/Personae/Pgroup/Persona
- //Play/Personae/Pgroup/Grpdescr

3.3.2 JFlap Usage

The diagram in figure 2 illustrates the operational procedures for converting an XML query to a DFA using JFlap.

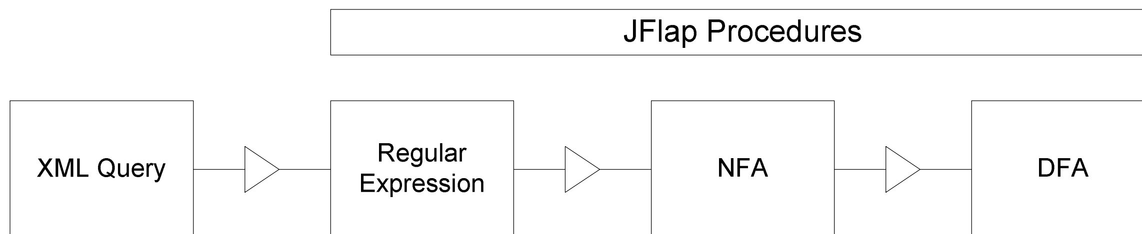

Figure 2. Operational Procedures to Convert XML query to a DFA using JFlap

The first operation is converting the XML query to a regular expression that JFlap can use. The JFlap regular expression tool only supports ASCII characters. JFlap converts each ASCII character into a separate state in the NFA. Thus, we represent each XML node and predicate with an ASCII character. In addition, a different ASCII character represents the end element for each node and predicate.

JFlap provides the instructions to convert the regular expression to an NFA, then to a DFA. We save the DFA as a JFlap file, which is an XML file. We count the states and transitions in the JFlap DFA saved file.

Example XML query → JFlap Regular Expression

```
//PLAY/ACT/TITLE
                  ELEMENT
NAME             START   END
-----            -----   -----
PLAY             a       z
ACT              b       y
TITLE            c       x
//               n*

// <PLAY><ACT><TITLE></TITLE></ACT></PLAY>
n*   a     b     c      x      y     z

//a/b/c
JFlap:
  : //a/b/c  -> n*abcxyz (JFlap regex)
```

3.3.3 Calculating post processing costs.

Calculating the post processing costs is the most complex mechanism used in our experiments. The post processing costs only apply to the joining of the sub patterns; it does not apply to the entire query. The diagram in figure 3 illustrates the process of collecting the post processing costs, ordered from left to right, of course.

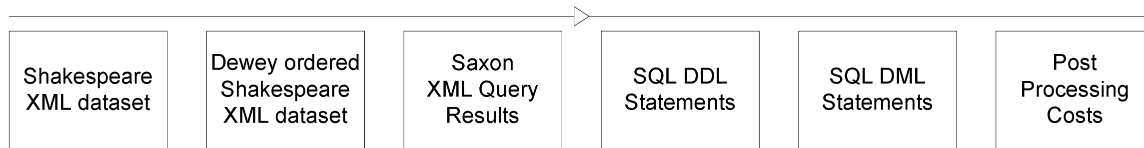

Figure 3. Post Processing Costs Procedures

As mentioned previously, we created a Java program (see Appendix) that modifies the original Shakespeare XML dataset to include an ‘id’ attribute. This ‘id’ uses a Dewey Ordering Encoding. “With Dewey Order, each node is assigned a vector that represents the path from the document’s root to the node. Each component of the path represents the local order of an ancestor node, as illustrated below.” [18]

```
Original Document             Dewey IDs          Modified Document
<a>                           1.                 <a id=”1.”>
  <b></b>                     1.1.                 <b id=”1.1.”></b>
  <c></c>                     1.2.                 <c id=”1.2.”></c>
</a>                                             </a>
<a>                           2.                 <a id=”2.”>
  <b></b>                     2.1.                 <b id=”2.1.”></b>
</a>                                             </a>
```

The next step involves using the Saxon XML parser to retrieve out intermittent results. The format of the query (stored as a .xsd file) and the result are illustrated below.

```
Query:
for $x in {XML Query}
return <entry><id>{data($x/@id)}</id><data>{data($x)}</data>

Results:
<entry>
  <id>{id value}</id>
  <data>{data}</data>
</entry>
```

We continue by passing the intermittent results of the XML query into a Perl script (see Appendix) we created on WPI’s Linux server. That script parses the data and creates the SQL DDL statements. We load the SQL DDL statements in Oracle where we can measure the post processing costs.

Finally, we measure the post processing costs by executing SQL DML statements. The SQL 'select' statement performs the structural joins that provide the post processing cost. The cost comparison is a join operation that compares the 'id' elements using the java procedure (see Appendix) xml_cmp_id. The following is an example of an SQL query that measures the post processing costs.

```
CREATE OR REPLACE VIEW q1_p12       # [query1, join of pattern 1 & 2] #
(ID) as
SELECT B.xml_id
FROM q1_pat1 A, q1_pat2 B
WHERE 1 = xml_cmp_id(A.xml_id, B.xml_id, 2)
GROUP BY B.xml_id;
```

4. Experimental Results

In this section we present our experimental results for the four queries provided in the paper. For each query we provide the breakdown of the XML query, the JFlap results, and the post processing costs.

4.1 XML Query 1

Query 1, Full: //Play/Act[Prologue]/Title

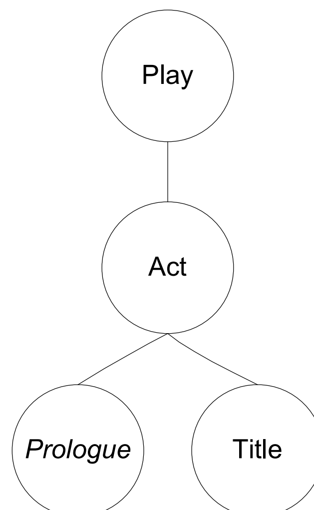

Figure 4. XML Tree for Query 1

4.1.1 Splitting the Query

Query 1, Pattern 1: //Play/Act/Prologue
Query 1, Pattern 2: //Play/Act/Title

4.1.2 JFlap Results

Here, we present the regular expressions used in the JFlap tool and then we provide the DFA node and transition count.

Regular Expression:

```
NAME            START   END
-----           -----   -----
Play            a       z
Act             b       y
Prologue        c       x
Title           d       w
//              n*

//a/b[c]/d

Pattern:
  : //a/b/c  -> n*abcxyz
  : //a/b/d  -> n*abdwyz

Full:
  : n*ab(cxdw+dwcx)yz
```

Node and Transition Count:

	Nickname	*Nodes*	*Transitions*
Query 1, Full	Q_1Full	14	16
Query 1, Pattern 1	Q_1P_1	8	9
Query 1, Pattern 2	Q_1P_2	8	9

4.1.3 Post Processing Costs

Here, we present two tables. The first table specifies the number of rows in the Oracle database for each of the sub-queries. The second table specifies the post processing cost when using Oracle. Each XML query contains these two tables.

Sub query	*Table entries*
Q_1P_1	12
Q_1P_2	185

Name	*Join*	*Time (seconds)*
Q_1T_1	$Q_1P_1 \bowtie Q_1P_2$	0.04

4.2 XML Query 2

Query 2, Full: //Play/Act[Title]/Scene[Title]/Stagedir

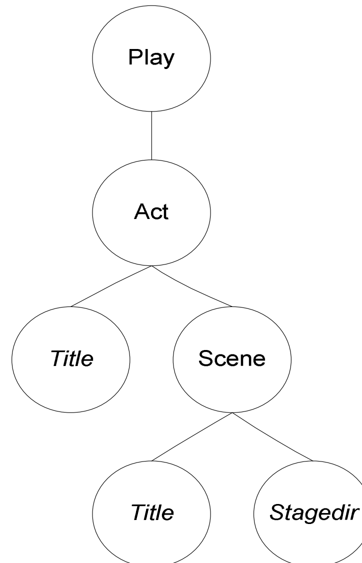

Figure 5. XML Tree for Query 2

4.2.1 Splitting the Query

Query 2, Pattern 1: //Play/Act/Title
Query 2, Pattern 2: //Play/Act/Scene/Title
Query 2, Pattern 3: //Play/Act/Scene/Stagedir

4.2.2 JFlap Results

Regular Expression:

```
NAME            START   END
-----           -----   -----
Play            a       z
Act             b       y
Title(a)        c       x
Scene           d       w
Title(s)        e       v
Stagedir        f       u
//              n*

//a/b[c]/d[e]/f

Pattern:
  : //a/b/c     -> n*abcxyz
  : //a/b/d/e   -> n*abdevwyz
  : //a/b/d/f   -> n*abdfuwyz

Full:
  : n*ab((cxd(evfu+fuev)w)+(d(evfu+fuev)wcx))yz
```

Node and Transition Count:

	Nickname	*Nodes*	*Transitions*
Query 2, Full	Q_2Full	80	84
Query 2, Pattern 1	Q_2P_1	8	9
Query 2, Pattern 2	Q_2P_2	22	23
Query 2, Pattern 3	Q_2P_3	22	23

4.2.3 Post Processing Costs

There are two ways in which we perform the structural join on the split queries. We join pattern 1 with pattern 3 (on the 'Act' portion of the id), and then join with pattern 2 ('Scene'). Or, we can join pattern 1 with pattern 2 ('Act'), and then join with pattern 3 ('Scene').

Sub query	*Table entries*
Q_2P_1	185
Q_2P_2	748
Q_2P_3	4251

Name	*Join*	*Time (seconds)*
Q_2T_1	$Q_2P_1 \bowtie Q_2P_3$	15.51
Q_2T_2	$Q_2T_1 \bowtie Q_2P_2$	73.53
Q_2T_3	$Q_2P_1 \bowtie Q_2P_2$	2.54
Q_2T_4	$Q_2T_3 \bowtie Q_2P_3$	61.44

4.3 XML Query 3

Query 3, Full: //Personae[Title]/Pgroup[Persona]/Grpdescr

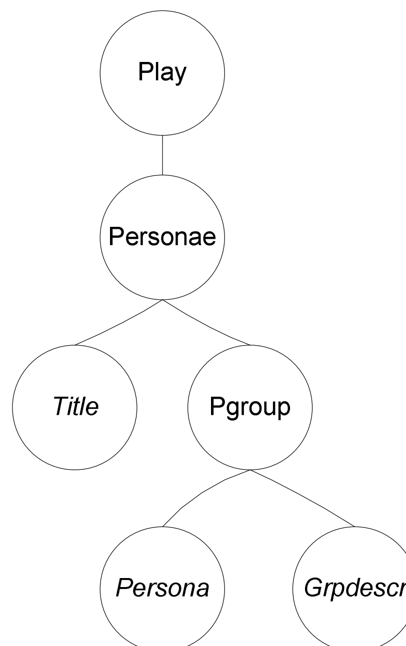

Figure 6. XML Tree for Query 3.

4.3.1 Splitting the Query

Query 3, Pattern 1: //Personae/Title
Query 3, Pattern 2: //Personae/Pgroup/Persona
Query 3, Pattern 3: //Personae/Pgroup/Grpdescr

4.3.2 JFlap Results

Regular Expression:

```
NAME            START   END
-----           -----   -----
Personae        a       z
Title           b       y
Pgroup          c       x
Persona         d       w
Grpdescr        e       v
//              n*

//a[b]/c[d]/e

Patterns:
  : //a/b       -> n*abyz
  : //a/c/d     -> n*acdwxz
  : //a/c/e     -> n*acevxz

Full:
  : n*a((by(cdwevx+cevdwx))+((cdwevx+cevdwx)by))
```

Node and Transition Count:

	Nickname	*Nodes*	*Transitions*
Query 3, Full	Q_3Full	82	86
Query 3, Pattern 1	Q_3P_1	14	15
Query 3, Pattern 2	Q_3P_2	18	19
Query 3, Pattern 3	Q_3P_3	18	19

4.3.3 Post Processing Costs

There are two ways in which we perform the structural join on the split queries. We join pattern 1 with pattern 3 (on the 'Personae' portion of the id), and then join with pattern 2 ('Pgroup'). Or, we can join pattern 1 with pattern 2 ('Personae'), and then join with pattern 3 ('Pgroup').

Sub query	*Table entries*
Q_3P_1	37
Q_3P_2	269
Q_3P_3	90

Name	*Join*	*Time (seconds)*
Q_3T_1	$Q_3P_1 \bowtie Q_3P_3$	0.07
Q_3T_2	$Q_3T_1 \bowtie Q_3P_2$	0.52
Q_3T_3	$Q_3P_1 \bowtie Q_3P_2$	0.19
Q_3T_4	$Q_3T_3 \bowtie Q_3P_3$	0.65

4.4 XML Query 4

Note: Figure 1, on page four illustrates the XML Tree for query 4.

Query 4, Full: //Play[Act[Title]/Scene[Title]/Stagedir]/ …
… Personae[Title]/Pgroup[Persona]/Grpdescr

4.4.1 Splitting the Query

Query 4, Pattern 1: //Play/Act/Title
Query 4, Pattern 2: //Play/Act/Scene/Title
Query 4, Pattern 3: //Play/Act/Scene/Stagedir
Query 4, Pattern 4: //Play/Personae/Title
Query 4, Pattern 5: //Play/Personae/Pgroup/Persona
Query 4, Pattern 6: //Play/Personae/Pgroup/Grpdescr

4.4.2 JFlap Results

Regular Expression:

```
NAME            START   END
-----           -----   -----
Play            a       z
Act             b       y
Title(a)        c       x
Scene           d       w
Title(s)        e       v
Stagedir        f       u
Personae        g       t
Title(p)        h       s
Pgroup          i       r
Persona         j       q
Grpdescr        k       p
//              n*

//a[b[c]/d[e]/f]/g[h]/i[j]/k

Patterns:
  : //a/b/c             -> n*abcxyz
  : //a/b/d/e           -> n*abdevwyz
  : //a/b/d/f           -> n*abdfuwyz
  : //a/g/h             -> n*aghstz
  : //a/g/i/j           -> n*agijqrtz
  : //a/g/i/k           -> n*agikprtz

Full:
  :
a(((b((cx(d(evfu+fuev)w))+((d(evfu+fuev)w)cx)y))(g((hs(i(jqkp+kpjq)r))+
((i(jqkp+kpjq)r)hs)t)))+((g((hs(i(jqkp+kpjq)r))+((i(jqkp+kpjq)r)hs)t))(
b((cx(d(evfu+fuev)w))+((d(evfu+fuev)w)cx)y))))

  : (b((cx(d(evfu+fuev)w))+((d(evfu+fuev)w)cx)y))
  : (g((hs(i(jqkp+kpjq)r))+((i(jqkp+kpjq)r)hs)t))
```

Node and Transition Count:

	Nickname	*Nodes*	*Transitions*
Query 4, Full	Q_4Full	322	334
Query 4, Pattern 1	Q_4P_1	8	9
Query 4, Pattern 2	Q_4P_2	22	23
Query 4, Pattern 3	Q_4P_3	22	23
Query 4, Pattern 4	Q_4P_4	14	15
Query 4, Pattern 5	Q_4P_5	18	19
Query 4, Pattern 6	Q_4P_6	18	19

4.4.3 Post Processing Costs

Query 4, as mentioned previously, combines queries 2 and 3. As such, the post processing costs from those queries are subparts of the post processing parts for this query. Thus, we use the lowest post processing costs from both queries to calculate the post processing costs for query 4. There are 4 ways in which we can perform the join for query 4. These join operations use the most efficient combination from both query 2 (Q_2T_4) and query 3 (Q_3T_2).

Sub query	*Table entries*
Query 4, Pattern 1	185
Query 4, Pattern 2	748
Query 4, Pattern 3	4251
Query 4, Pattern 4	37
Query 4, Pattern 5	269
Query 4, Pattern 6	90

Name	*Join*	*Time (seconds)*
Q_4T_1	$Q_2T_4 \bowtie Q_3T_2$	61.44
Q_4T_2	$Q_2Full \bowtie Q_3T_2$	7.63
Q_4T_3	$Q_2T_4 \bowtie Q_3Full$	300+
Q_4T_4	$Q_2Full \bowtie Q_3Full$	7.04

5. Analysis and Conclusion

Our goal is to determine the most efficient approach for evaluating an XML query. As mentioned throughout the paper, the query or its sub-patterns can be satisfied inside the automaton or outside the automaton. The following table summarizes the costs associated with processing an XML query.

	Nodes + Transitions	*Post Processing (seconds)*
Query 1		
Q_1Full	30	None
Patterns	34	0.4
Query 2		
Q_2Full	164	None
Most efficient join	127	61.44
Query 3		
Q_3Full	164	None
Most efficient join	127	0.52
Query 4		
Q_4Full	656	None
$Q_2T_4 \bowtie Q_3T_2$	254	62.51
Q_2Full $\bowtie$ Q_3T_2	291	7.63
$Q_2T_4 \bowtie$ Q_3Full	291	300+
Q_2Full $\bowtie$ Q_3Full	328	7.04

Analyzing the results of the table above, the most significant observations pertain to the results of query 4, as it is the combination of queries 2 and 3. Performing the query exclusively using a DFA requires 656 states and transitions. That will require a considerable amount of memory compared to the other methods for resolving query 4. When query 4 is split into queries 2 and 3, the number of states in the DFA is cut in half. However, there is a large difference between the possible join combinations.

For instance, Q_2Full $\bowtie$ Q_3T_2 requires less post processing cost than $Q_2T_4 \bowtie$ Q_3Full even though both have the same number of states and transitions. An optimizer examining these statistics might choose Q_2Full $\bowtie$ Q_3T_2 or Q_2Full $\bowtie$ Q_3Full over the other three options.

Moreover, digging deeper into query 2 and query 3 shows another important result related to the number of tuples in the table. When joining the sub patterns in both queries 2 and 3, the operation is faster when the join occurs between the tables with the least amount of elements first. Thus, the size of the tuples returned by patterns impact the post processing cost. Also worth noting is that we did not notice any benefit in using an index for any of the tables, though this may not be true in all cases.

Finally, we propose an overall architecture for the optimizer. The optimizer takes the XML query as input and it decides which option is best for solving the query. The options that the optimizer must take into account are the query and data statistics, the estimated post processing cost, the size of the automaton, and the time for matching. In our research, we only investigate the post processing cost and the size of the automaton. Also note, our methodology utilizes both approaches (inside/outside automaton) for solving the XML query. The other parts of the optimizer require future work, as discussed below.

Determining how and where to split the XML query is a topic for further investigation. In our experiments, we use the Raindrop approach [10]. Additional experiments could help define a baseline for splitting the query. This baseline could take into account the number of states and transitions in the DFA and the post processing costs (estimated from number of rows). This leads us to the next possible research topic; estimating the number of tuples that could match a given sub pattern of an XML query.

Lastly, our investigation only pertains to NFA/DFAs. We did not explore PDAs, which are more general than NFA/DFAs. PDAs can handle both non-recursive and recursive XML documents [19]. PDAs will require additional space in the finite automaton because it also keeps a stack. However, the PDA can perform more of the structural join operations within the automaton using the stack. This will reduce the post processing costs and measuring that cost is also a future topic of investigation.

Appendix

Here we provide the source and script code used in our project. There are three components. The first is the Java program that inserts the Dewey Ordering id into the Shakespeare XML dataset. The second is the comparison routine used in Oracle. Last is the Perl script that converts the Saxon XML query results to SQL DDL statements.

Java Program (pages 18 - 21)
This program uses the SAX (Simple API for XML) Java package to parse the Shakespeare XML dataset. The Java class 'MySAXApp' is a modification of the quick start tutorial provided by the SAX Project [17]. This class contains a main() function and runs as a standalone program. The class takes an argument, which should be an XML file. If it is not an XML file, the SAX package will throw an exception. We name the modified document with the Dewey ids 'testing.xml'. We attempt to maintain the integrity of the XML file, though, there are output issues pertaining to escaped characters. We fixed these issues manually.

Algorithm: When a start element event occurs, increment the document depth and increment the counter for that depth. When an end element event occurs, decrement the document depth and reset the value of the counter for depth + 2. Using the example from page 6, the following table illustrates this algorithm.

	Element	Document Depth				Dewey ID
		0	**1**	**2**	**3**	
Start Document		→ 0	0	0	0	
Start Element	A	0	→ **1**	0	0	1.
Start Element	B	0	1	→ **1**	0	1.1
End Element	B	0	→ 1	1	□ 0	
Start Element	C	0	1	→ **2**	0	1.2
End Element	C	0	→ 1	2	□ 0	
End Element	A	→ 0	1	□ 0	0	
Start Element	A	0	→ **2**	0	0	2.
Start Element	B	0	2	→ **1**	0	2.1
End Element	B	0	→ 2	1	□ 0	
End Element	A	→ 0	2	□ 0	0	
End Document						

Key
→ : current depth
□ : number reset

Oracle Java Procedure (page 22)
When performing the structural joins in Oracle, we compare the Dewey ids for the given tuples. The comparison of the ids pertains to the level of the join. For example, joining the tuples with ids 1.2.1 and 10.2.4 respectively at level 2 is true. Joining the same two tuples at level 1 is false.

Oracle provides functions such as substr [13] that facilitate this string comparison. However, our comparison only pertains to the level of the join, and the Dewey ids may have different string lengths. Thus, using the substr is not an optimal solution. Therefore, we created a java function that we incorporate into Oracle.

The function xml_cmp_id takes three parameters; the first two are the Dewey ids for the comparison, and the third is the level the comparison will take place. The function returns 1 if successful, otherwise it returns 0.

Perl Script (pages 23 – 24)
We created a Perl script that converts the output of the Saxon XML parser to SQL DDL statements. The script uses basic regular expressions to extract the data and format the DDL statements. It requires one argument; the prefix of the filename. A second option allows the user to specify an index for the table. The file must end with '.xml'. The script creates two SQL DDL files; one to create to create the table definition (.ctl) and the other for inserting the data into the table (.sql).

For example, if we execute the script passing in 'test' as the only argument, the script will read the file 'test.xml'. The script will create two files; 'test.ctl' and 'test.sql'. 'test.ctl' contains the table definition, while 'test.sql' contains the 'insert into' definitions.

Java Version Information in Eclipse [12]:

Version: 3.4.1
Build id: M20080911-1700

java version "1.6.0_07"
Java(TM) SE Runtime Environment (build 1.6.0_07-b06)
Java HotSpot(TM) Server VM (build 10.0-b23, mixed mode)

Java Program

```
import java.io.*;
import org.xml.sax.XMLReader;
import org.xml.sax.Attributes;
import org.xml.sax.InputSource;
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.helpers.DefaultHandler;

public class MySAXApp extends DefaultHandler {

    final  int XPATH_DEPTH = 20;
    public int currentDepth;
    public int depthId[];
    public int sElem = 0;
    public int eElem = 0;
    public FileOutputStream fos;

    public void init() { return; }

    public MySAXApp() {
        super();

        depthId = new int [XPATH_DEPTH];
        for (int i=0; i < XPATH_DEPTH; i++) {
            depthId[i] = 0;
        }
        currentDepth = 0;

        // open up the output file
        try {
            fos = new FileOutputStream("testing.xml");
        }
        catch (IOException e) {
            System.err.println( e.toString() + "File not found");
        }
    }

    /**
     * @param args
     */
    public static void main(String[] args)
    throws Exception
    {
        XMLReader xr = XMLReaderFactory.createXMLReader();
        MySAXApp handler = new MySAXApp();
        xr.setContentHandler(handler);
        xr.setErrorHandler(handler);

        // Parse each file provided on the command line
        for (int i = 0; i < args.length; i++) {
            System.out.println(args[i]);
            FileReader r = new FileReader(args[i]);
            xr.parse(new InputSource(r));
        }
    }
```

```
        ////////////////////////////////////////////////////////////////////
        // Event handlers.
        ////////////////////////////////////////////////////////////////////

        public void startDocument () {
            String xmlVersion = "<?xml version=\"1.0\"?>\n<!DOCTYPE PLAY SYSTEM
    \"play.dtd\"\n";
            System.out.println(xmlVersion);
            try {
                fos.write(xmlVersion.getBytes());
            }
            catch (IOException e) {
                System.err.println( e.toString() + "File not found");
            }
        }


        public void endDocument () {
            //System.out.println("End document");
}

public void startElement (String uri, String name, String qName, Attributes atts) {
    // Increase level, and update the counter
    currentDepth++;
    depthId[currentDepth]++;

    // Create the output buffer
    StringBuffer outBuf = new StringBuffer();

    // Build the string
    // 1. Should there be an enter?
    if (sElem > 0) {
        outBuf.append("\n");
    }
    // 2. What is the offset
    for (int joe=1; joe < currentDepth; joe++) {
        outBuf.append("  ");
    }
    // 3. Start of the element
    outBuf.append("<"); outBuf.append(qName); outBuf.append(" id=\"");
    // 4. Build the id portion
    for (int i=1; i <= currentDepth; i++) {
        outBuf.append(depthId[i]); outBuf.append(".");
    }
    // 5. Tack on the end
    outBuf.append("\">");

    String outStr = new String(outBuf);
    // 6. Write to the screen and the file
    System.out.println(outStr);
    try {
        fos.write(outStr.getBytes());
    }
    catch (IOException e) {
        System.err.println( e.toString() + "File not found");
    }
```

```
        // Update some display control elements
        sElem++;
        eElem = 0;
    }

    public void endElement (String uri, String name, String qName) {
        // Decrease level, reset the count for depth id of
        // current depth level + 2
        currentDepth--;
        depthId[currentDepth+2] = 0;

        // Create the output buffer
        StringBuffer outBuf = new StringBuffer();

        // Build the string
        // 1. Create the offset
        if (eElem > 0) {
            for (int joe=0; joe < currentDepth; joe++) {
                outBuf.append("  ");
            }
        }
        // 2. Add the tag
        outBuf.append("</"); outBuf.append(qName); outBuf.append(">\n");

        String outStr = new String(outBuf);
        // 3. Write to the screen and file
        System.out.print(outStr);
        try {
            fos.write(outStr.getBytes());
        }
        catch (IOException e) {
            System.err.println( e.toString() + "File not found");
        }

        // Update some display control elements
        eElem++;
        sElem = 0;
    }

    public void characters (char ch[], int start, int length) {
        //System.out.print("Characters...: \"");
        for (int i = start; i < start + length; i++) {
            switch (ch[i]) {
            case '\\':
                System.out.print("\\\\");
                try {
                    fos.write('\\'); fos.write('\\');
                }
                catch (IOException e) {
                    System.err.println( e.toString() + "File not found");
                }
                break;
            case '"':
                System.out.print("\\\"");
                try {
                    fos.write('\"');
```

```
                }
                catch (IOException e) {
                    System.err.println( e.toString() + "File not found");
                }
                break;
            case '\n':
                System.out.print("\n");
                try {
                    fos.write('\n');
                }
                catch (IOException e) {
                    System.err.println( e.toString() + "File not found");
                }
                break;
            case '\r':
                System.out.print("\\r");
                try {
                    fos.write('\r');
                }
                catch (IOException e) {
                    System.err.println( e.toString() + "File not found");
                }
                break;
            case '\t':
                System.out.print("\\t");
                try {
                    fos.write('\t');
                }
                catch (IOException e) {
                    System.err.println( e.toString() + "File not found");
                }
                break;
            case '&':
                System.out.print("&amp;");
                try {
      fos.write('&');fos.write('a');fos.write('m');fos.write('p');fos.write(';');
                }
                catch (IOException e) {
                    System.err.println( e.toString() + "File not found");
                }
                break;
            default:
                System.out.print(ch[i]);
                try {
                    fos.write(ch[i]);
                }
                catch (IOException e) {
                    System.err.println( e.toString() + "File not found");
                }
            break;
            }
        }
        //System.out.print("\"\n");
    }
}
```

Oracle Java Procedure

```
public class SqlXmlCmp {

    public static int
        xml_cmp_id(java.lang.String s1,
                   java.lang.String s2,
                   int lvl)
    {
        int i;
        int j = 0;
        int length = s1.length();

        for (i=0; i < length; i++) {
            if (s1.charAt(i) != s2.charAt(i)) {
                return (0);
            }
            // Perform this check after comparison
            if (s1.charAt(i) == '.') {
                j++;
                if (j == lvl) {
                    return (1);
                }
            }
        }
        return (1);
    }
}

CREATE OR REPLACE FUNCTION xml_cmp_id (s1 VARCHAR2,
                                       s2 VARCHAR2,
                                       lvl NUMBER)
RETURN NUMBER AS LANGUAGE JAVA
NAME 'SqlXmlCmp.xml_cmp_id(java.lang.String, java.lang.String, int)
                           return int';
/
```

Perl Script

```
#!/usr/bin/perl -w

use strict;

# Command line options
my $prefix = shift(@ARGV);
my $indexName = shift(@ARGV);

# specify the other information
my $xmlFile = "$prefix.xml";
my $sqlFile = "sql/$prefix.sql";
my $ctlFile = "sql/$prefix.ctl";

# set the table name
my $tableName = "$prefix";
$tableName =~ s/\-/\_/;

# vairables for sql table output
my $entryId; my $entryData;

# variable for debugging
my $lineCount = 0;

# open xml file to read
open(my $fd, "<", $xmlFile) or die "Could not read $xmlFile";
# open sql (table create) and ctl (data insert) file to write
open(my $sqlFd, ">", $sqlFile) or die "Could not open $sqlFile for writing";
open(my $ctlFd, ">", $ctlFile) or die "Could not open $ctlFile for writing";

#write the .sql file
print $sqlFd "DROP TABLE $tableName cascade constraints;\n";
print $sqlFd "CREATE TABLE $tableName (\n";
print $sqlFd "xml_id VARCHAR(32),\n";
print $sqlFd "xml_data VARCHAR(255)\n";
if ($indexName) {
   print $sqlFd ", CONSTRAINT pk_$tableName PRIMARY KEY $indexName)\n";
   # CONSTRAINT pk_owner PRIMARY KEY (own_id)
}
print $sqlFd ");\n\n";

# write the .ctl header
print $ctlFd "LOAD DATA\n";
print $ctlFd "INFILE *\n";
print $ctlFd "INTO TABLE $tableName\n";
print $ctlFd "FIELDS TERMINATED BY \"::::\"\n";
print $ctlFd "( xml_id, xml_data )\n\n";

# write the .ctl data section
print $ctlFd "BEGINDATA\n";
while (my $line = <$fd>) {
    chomp $line;
    $lineCount++;

    # determine what we have
```

```
    if ($line =~ /^\<entry\>/ ) {
        # Need to reset all values
        $entryId = $entryData = $entrySid = $entryFid = $entryLvl = 0;
    } elsif ($line =~ /^\<\/entry\>/) {
        # Need to print
        if ($tableType == 1) {
            print $ctlFd "$entryId :::: $entryData\n";
        } elsif ($tableType == 2) {
            print $ctlFd "$entrySid :::: $entryFid :::: $entryLvl ::::
$entryData\n";
        }
    } elsif ($line =~ /\<id\>(.*)\<\/id\>/) {
        $entryId = $1;
    } elsif ($line =~ /\<data\>(.*)\<\/data\>/) {
        $entryData = $1;
    } else {
        # do nothing
    }
}
#print $ctlFd "ENDDATA\n\n";

print "Line Count: $lineCount\n";

close $fd; close $sqlFd; close $ctlFd;
```

References

[1] S.S. Chawathe and F. Peng. XPath Queries on Streaming Data. In the ACM SIGMOD Conference, pages 431-442, 2003.
[2] M. Li, M. Mani, E.A. Rundensteiner, H. Su, and M. Wei. XML Stream Processing: Current Technologies and Open Challenges. Worcester Polytechnic Institute, NSF Grant No. IIS-0414567.
[3] T.J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. International Conference on Database Theory, pages 173-189, 2003.
[4] A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In the ACM SIGMOD Conference, pages 419-430, 2003.
[5] B. Ludascher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. International Conference on Very Large Databases, pages 227-238, 2002.
[6] Y. Diao, M. Altinel, M.J. Franklin, H. Zhang, and P. Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. ACM Transactions on Database Systems, pages 467-516, Vol. 28, No.4, December 2003.
[7] G. Agrawal and X. Li. Efficient Evaluation of XQuery over Streaming Data. International Conference on Very Large Databases, pages 265-276, 2005.
[8] Y. Chen, S.B. Davidson, and Y. Zheng. An Efficient XPath Query Processor for XML Streams. Proceedings of the 22nd International Conference of Data Engineering, 2006. IEEE Computer Society.
[9] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. International Conference on Very Large Databases, pages 228-239, 2004.
[10] M. Li, M. Mani, E.A. Rundensteiner, and M. Wei. Processing recursive XQuery over XML streams: The Raindrop approach. Data and Knowledge Engineering 65, pages 243-265, 2008.
[11] SAXON: The XSLT and XQuery Processor. Version 9.1.0.2. http://saxon.sourceforge.net/
[12] Eclipse Software Development Kit. Version . http://www.eclipse.org/
[13] WPI Oracle Database. Oracle Database 10g Enterprise Edition Release 10.2.0.3.0 - 64bit Production. http://www.oracle.com
[14] JFlap Version 6.4 (July 13th, 2008). http://www.jflap.org/
[15] J. Bosak. Shakespeare XML Dataset. Version 2.0. http://xml.coverpages.org/bosakShakespeare200.html
[16] J. Clark and S. DeRose. XML Path Language (XPath). Version 1.0. http://www.w3.org/TR/xpath
[17] SAX Project Quickstart. http://www.saxproject.org/quickstart.html
[18] J. Beyer, E. Shanmugasun-daram, J. Shekita, S. Tatarinov, K.S. Viglas, and C. Zhang. Storing and querying ordered XML using a relation database system. SIGMOD Conference, pages 204-215, 2002.
[19] L. Segoufin and V. Vianu. Validating Streaming XML Documents. ACM PODS, pages 53-64, 2002.