CS276B Project Report: Streaming XPath Engine

Amruta Joshi Oleg Slezberg

ABSTRACT

XPath [1] has received a lot of attention in research during the last five years. It is widely used both as a standalone language and as a core component of XQuery [2] and XSLT [3]. Our project (titled xstream) concentrated on evaluation of XPath over XML streams. This research area contains multiple challenges resulting from both the richness of the language and the requirement of having only a single pass over the data. We modified and extended one of the known algorithms, TurboXPath [4], a tree-based IBM algorithm. We also provide extensive comparative analysis between TurboXPath and XSQ [5], currently the most advanced of finite automata (FA)-based algorithms.

1. INTRODUCTION

Querying XML streams has a variety of applications. In some, data occurs naturally in the streaming form (e.g. stock quotes, news feeds, XML routing). In others, streaming is beneficial for the performance reasons, since the data is accessed using a single sequential scan. Whatever the reason, evaluation of XML streams using XPath poses many challenges. First, the evaluation must be done in one pass over the data. Secondly, XPath is a rich functional language, whose implementation is a nontrivial task. In addition to that, many XPath features such as descendant axis, predicate evaluation, and wildcards require elaborate algorithms in order to be processed efficiently.

A generic XPath query can be represented as a sequence of *location steps*, where each step contains *axis*, *node test*, and zero or more *predicates* associated with it. A last location step is called an *output expression*; this is the expression that answers the query. For example, in a query //book[price < 30]/title we are querying for titles of all books priced under $30. The location steps here are //book[price < 30] (axis is //, node test is book predicate is [price < 30]) and title (axis is /, node test is title, no predicate). The output expression here is title.

2. RELATED WORK

At the dawn of the XPath streaming evaluation, the researchers tried to solve the problem by building a finite automaton out of XPath query and running the SAX parser events through this automaton. Different forms of FA were tried – DFA [14] and NFA [15], built during preprocessing or lazily. The most recent of the FA algorithms is XSQ. It uses a sophisticated XPDT (eXtended PushDown Transducer) model for its hierarchical automaton, due to which it claims to provide the highest level of the XPath language support.

On the other hand, TurboXPath is a tree-based XPath streaming algorithm. It first builds parse tree (PT) for the input query and then keeps matching the streamed document against the PT nodes. It uses smart matching arrays during the matching, avoiding exponential memory usage typical for FA algorithms.

The name 'TurboXPath' is associated with several versions of the algorithm. It was first used in [10], then in [9], and finally in [4]. We use version [4] in our project since it is the most recent of all of the above.

3. PRELIMINARIES

3.1. General Implementation Information

We implemented TurboXPath in Java using JDK 1.4.2. All our testing was done on Solaris 2.8. We used Apache Xerces 2.6.2 [6] for both XPath and XML parsers. Since XSQ implementation is also in Java, this allowed us to provide a more precise performance comparison between the implementations. The earlier benchmarks in [4] favor TurboXPath based on a comparison between its C++ and XSQ Java implementations.

3.2. Benchmark

We chose XPathMark [7] as our benchmark for both functional and performance analysis. Note that the link [7] has changed after we started our project, so we use a slightly older version containing 23 queries. See Appendix 2 for the full list of queries.

The benchmark comes with an XML document, containing the data to run the queries on and the file of answers. The document contains 60K of data (785 elements, depth = 11) modeling an Internet auction Web site.

Our first intention was to use XMark [8] for our analysis. However, the investigation showed that XMark is primarily an XQuery benchmark and does not provide an adequate XPath coverage. On other hand, XPathMark exercises a broad variety of XPath axes, predicates, and functions.

As we will see later in the document, XPathMark is a very predicate-intensive benchmark, whose 73% of the queries contain predicates. We do see it as a plus. In the real world, the queries will likely be complex, since they will likely be generated by application software and not typed by an end-user. Hence having higher level of complexity in the benchmark makes it closer to the real-life applications.

4. TURBOXPATH ANALYSIS

4.1. Algorithm corrections

We started with the following pseudo-code from [4]:

```
Function startElement(x)
1: maxIndex := nextIndex - 1
2: for i := 0 to maxIndex do
3:    u := pointerArray[i]
4:    if ((ntest(u) = label(x)) or (ntest(u) = *)) then
5:          if ((axis(u) = descendant) or
6:             (levelArray[i] = currentLevel)) then
7:                if (not validationArray[i]) then
8:                      for c in children(u) do
9:                            if ((c = firstSibling(c)) and
10:                           (recursionArray[u] = 0)) then
11:                               pointerArray[i] := c
12:                               levelArray[i]:=currentLevel+1
13:                           else
14:                               pointerArray[nextIndex] := c
15:                               levelArray[nextIndex]
                                       :=currentLevel+1
16:                               validationArray[nextIndex]:=0
17:                               nextIndex := nextIndex + 1
18:   recursionArray[u]:=recursionArray[u]+1
19: currentLevel := currentLevel + 1
```

```
Function endElement(x)
1: currentLevel := currentLevel - 1
2: i := nextIndex - 1
3: while (levelArray[i] > currentLevel) do
4:          u := pointerArray[i]
5:          if ((u = firstSibling(u) and
6:                (recursionArray[parent(u)] = 0)) then
7:                pointerArray[i] := parent(u)
8:                levelArray[i] := currentLevel
9:          else
10:               nextIndex := nextIndex - 1
11:         i := i - 1
12: for i := 0 to (nextIndex - 1) do
13:    u := pointerArray[i]
14:    if ((ntest(u) = label(x)) or (ntest(u) = *)) then
15:         if ((axis(u) = descendant) or
16:               (levelArray[i] = currentLevel)) then
17:               if (isLeaf(u)) then
18:                   validationArray[i] := evalPred(u)
19:               else
20:                   c:=validationArray bits for children(u)
21:                   if (evalPred(c)) then
                         validationArray[i]:=true
22:                   recursionArray[u]:=recursionArray[u]-1
23:               if (u = $) then
24:                     queryResponse := validationArray[i]
```

Here is the explanation of the different structures and variables:

1. x – current element name.
2. pointerArray – array of nodes being matched.
3. validationArray – array remembering if these nodes were ever matched.
4. nextIndex – index of the next available slot in pointerArray and validationArray.
5. maxIndex – max node index.
6. i – just an iteration variable.
7. ntest(u), axis(u) – a PT node test/axis.
8. label(x) – current element name.
9. levelArray[i] – level (depth) at which to match the node.
10. currentLevel – current depth.
11. recursionArray[u] – level of recursion of the node u (i.e. how many times we've observed <u> inside <u>).
12. queryResponse – algorithm result, whether there is a query match or not.

We discovered and fixed the following issues with this algorithm.

4.1.1. Closing child elements

Suppose that we have a simple child-only query, e.g. /a/b/c. For each matching startElement(), we will be executing line 11 that replaces the parent PT node with its first (and in our case only) child. Then, when this element closes, in endElement(), we will need to do the reverse procedure: replace the child node with its parent (lines 7-8). Unfortunately, this will never happen as no node will ever satisfy the condition on line 6, since we unconditionally increment the open element recursion count on line 18 of startElement().

The solution to this is not to bump up the recursion level for child nodes, only for descendants. The intuition behind this is that we stop matching child nodes (unlike descendants!) after the original match, so there is no reason to worry about their recursion count. The check for descendant axes must be placed both when incrementing the count (line 18 of startElement()) and when decrementing it (endElement(), line 22).

4.1.2. Recursive documents

A document d is *recursive* with respect to a query q, if there exists a node n in the parse tree of q and two elements e1, e2 in d, such that:

1. Both e1 and e2 match n.
2. e1 contains e2.

For example, the document below is recursive with respect to query //a, but is not recursive with respect to query //b.

```
<a>
    <a>
        <b/>
    </a>
    <b>
    </b>
</a>
```

Though this might sound like too rare of a thing to worry about, we observe that *any* document is recursive with respect to query //*. Also, be aware that a survey of 60 **real** datasets found 35 (58%) to be recursive [13].

Recursive documents put a lot of strain on streaming XPath processors as you need to keep matching n against both e1 and e2 when you go inside e2. This results into exponential number of states for FA-based algorithms. As first shown in [9], things are better for the tree-based algorithms. For example, suppose our query is //a/b. Then, in the startElement() code above, when we enter the outer instance of a, we go inside the if on line 10, to lines 11 and 12, and ***replace*** a with b inside the array of nodes to match. Therefore, the inner instance of a will never be matched.

The solution to this is to move recursionArray[u] increment on line 18 upwards, to precede the check on line 10. Symmetrically, we will need to also move a corresponding decrement in endElement(). The new descendant conditions we introduced in Closing child elements will need to be moved too.

4.1.3. Array bounds

Another small change we made was in endElement(), line 3: We need to check (i >= 0) in order not to cross the array boundary.

4.1.4. Corrected version

Applying all corrections algorithm is (changes in **bold**):

```
Function startElement(x)
1: maxIndex := nextIndex - 1
2: for i := 0 to maxIndex do
3:      u := pointerArray[i]
4:      if ((ntest(u) = label(x)) or (ntest(u) = *)) then
5:          if ((axis(u) = descendant) or
6:            (levelArray[i] = currentLevel)) then
              if (axis(u) == descendant) then
                  recursionArray[u]++;
7:            if (not validationArray[i]) then
8:                for c in children(u) do
9:                    if ((c = firstSibling(c)) and
10:                   (recursionArray[u] = 0))then
11:                       pointerArray[i] := c
12:                       levelArray[i]
                            :=currentLevel+1
13:                   else
14:                       pointerArray[nextIndex] := c
15:                       levelArray[nextIndex]
                            :=currentLevel + 1
16:                       validationArray[nextIndex] := 0
17:                       nextIndex := nextIndex + 1
~~18:     recursionArray[u]:=recursionArray[u]+1~~
19: currentLevel := currentLevel + 1
```

```
Function endElement(x)
1: currentLevel := currentLevel - 1
2: i := nextIndex - 1
3: while ((i >= 0) and (levelArray[i] > currentLevel))do
4:      u := pointerArray[i]
5:      if ((u = firstSibling(u) and
6:          (recursionArray[parent(u)] = 0)) then
7:          pointerArray[i] := parent(u)
8:          levelArray[i] := currentLevel
9:      else
10:         nextIndex := nextIndex - 1
11:     i := i - 1
12: for i := 0 to (nextIndex - 1) do
13:   u := pointerArray[i]
14:   if ((ntest(u) = label(x)) or (ntest(u) = *)) then
15:       if ((axis(u) = descendant) or
16:           (levelArray[i] = currentLevel)) then
            if (axis(u) == descendant) then
                recursionArray[u]--;
17:         if (isLeaf(u)) then
18:             validationArray[i] := evalPred(u)
19:         else
20:             c:=validationArray bits for Children(u)
21:             if(evalPred(c)) then
                  validationArray[i] := true
~~22:     recursionArray[u]:=recursionArray[u]+1~~
23:         if (u = $) then
24:             queryResponse:=validationArray[i]
```

4.2. *Evaluation extension*

There are two distinct questions that can be asked while processing XPath query against a streaming document:

1. Does this document match the query?
   F1: XML => Boolean

2. What parts of the document match the query?
   F2: XML => XML

The problem F1 is called XPath *filtering*. F2 is known as XPath *evaluation*. Our project implements F2. Evaluation is a harder problem than filtering. An evaluation algorithm can be easily converted into a filtering algorithm (by comparing its result to the empty set), but not vice versa.

Since [4] describes only a filtering algorithm, we had to extend it to do evaluation. This was done by implementing the following extensions:

1. Output buffers for predicate owner nodes (i.e. nodes that have predicates associated with them). We cannot output them as we stream because the predicates can only be evaluated on the element close tag. For this, we adopted the model explained in [9].

2. Predicate node buffers. This will buffer contents of nodes that are used inside the predicate expressions. This also was explored in [9].

3. Predicate evaluation. This was omitted from [4] due to lack of space; however, manipulation of multiple predicate buffers, joins between them, and propagation of the evaluation result are not trivial.

In addition, we had to redo the validationArray updations (e.g. remove startElement(), line 7) to adapt it to the needs of an evaluation algorithm.

4.3. Multiple expressions

There is one other thing that we noticed working on the TurboXPath implementation. In [9], the authors present TurboXPath as not capable of evaluating multiple XPath expressions simultaneously on the same XML stream. This is something that was successfully done in many earlier algorithms, e.g. [11] and [12]. However, since TurboXPath is capable of processing multiple output nodes, we can just combine the queries, OR-ing them together:

$q = (q_1) \mid (q_2) \mid \ldots \mid (q_n);$

which produces a single query that can be handled by TurboXPath. Evaluating q will achieve the effect of evaluating $(q_{1,}$ q2, …, $q_n)$ simultaneously. Note that eliminating common sub-expressions on q (as a post-

Query	**XSQ result**	**XSQ failure reason**	**xstream result**	**Xstream failure reason**
Q1	fail	* is unsupported.	pass	n/a
Q2	pass	n/a	pass	n/a
Q3	pass	n/a	pass	n/a
Q4	pass	n/a	pass	n/a
Q5	fail	Backward axes unsupported.	fail	Transforms into a DAG query, these are unsupported (see Backward axis processing for more details).
Q6	fail	Backward axes unsupported.	fail	Transforms into a recursive query w/predicate, these are unsupported (unlike recursive queries w/o predicates).
Q7	fail	Following axis unsupported.	fail	Following axis is unsupported.
Q8	fail	Preceding axis unsupported.	fail	Preceding axis is unsupported.
Q9	fail	Following axis unsupported.	fail	Following axis is unsupported.
Q10	fail	Preceding axis unsupported.	fail	Preceding axis is unsupported.
Q11	fail	Functions are unsupported.	fail	DTDs are unsupported.
Q12	fail	Functions are unsupported.	fail	DTDs are unsupported.
Q13	fail	Functions are unsupported.	fail	DTDs are unsupported.
Q14	fail	Functions are unsupported.	fail	DTDs are unsupported.
Q15	fail	Functions are unsupported.	fail	DTDs are unsupported.
Q16	fail	"And" predicates unsupported.	fail	Multivariate (> 1 location step) predicates are unsupported.
Q17	fail	Functions are unsupported.	fail	DTDs are unsupported.
Q18	fail	Functions are unsupported.	fail	Positional predicates are unsupported.
Q19	fail	Functions are unsupported.	pass	n/a
Q20	fail	A bug, does not work for string constants with spaces.	pass	n/a
Q21	fail	* is unsupported.	pass	n/a
Q22	pass	n/a	pass	n/a
Q23	fail	Functions are unsupported.	fail	DTDs are unsupported.
#passed (of 23)	4		8	
Coverage (%)	17%		34%	

processing) will optimize the PT. This is not mandatory for the TurboXPath to work though. Note that this is a purely theoretical contribution.

5. COMPARATIVE ANALYSIS

A big part of our project was comparing our implementation with XSQ, a newest and the most advanced of FA-based algorithms. We used XPathMark, described in section 2.3, for both functional and performance analysis.

5.1. Functionality

The functional coverage analysis were performed by running XPathMark queries and verifying the output correctness by comparing it to the correct answers (supplied as a part of the benchmark).

Table on previous page shows the results for both the implementations and explanations for possible failures. Note that, in XSQ case, our reasons for failure might not be 100% accurate, they are based on our observations. One of the reasons for XSQ low coverage is the fact that 17 out of the 23 XPathMark queries have predicates. XSQ processes correctly only one of these queries (Q22).

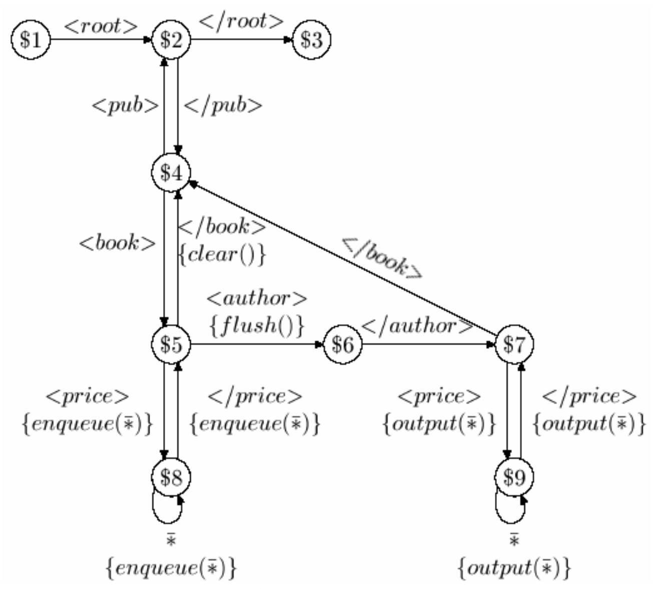


Figure 1: XSQ XDPT for /pub/book[author]/price

Let’s investigate the reasons behind the inadequate XSQ predicate support.

5.1.1. Predicate evaluation

In order to illustrate how FA and tree-based predicate evaluation works, let’s take the following query from [5]: /pub/book[author]/price.

The XSQ XPDT is is shown on Figure 1. The corresponding TurboXPath PT can be found on figure 2. Note that in PT an arrow denotes a predicate dependency; double circle – an output node.

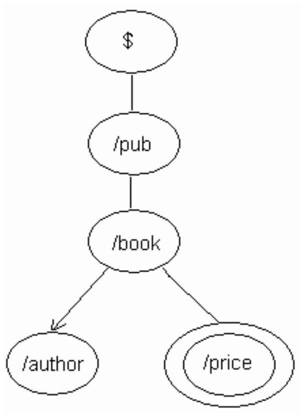


Figure 2: TurboXPath PT for /pub/book[author]/price

The figures look very similar. In fact, if you eliminate close tag XPDT transitions and the output queue actions, they will look almost identical. With one notable exception, though.

Queries with predicates (e.g. author in our example) increase the number of children of the predicate owner node (in our example, book). In this example, we keep track of 2 book’s children, author and price; this can potentially be any number of child nodes.

Moreover, we will need to keep track of all permutations of the children. Note how from XPDT state 6 we have a transition to state 7 on </author>. And, in state 7, we start matching <price> again, just as we tried in state 5.

In a general case, if a node has m children, you would need m! states/transitions to handle the situation.

On other hand, TurboXPath avoids this by using the smart matching array logic described in the TurboXPath analysis section. While FA-based algorithms can potentially add O(m!) transitions to take care of the predicate owner node children, TurboXPath handles it with m entries in the matching array.

So while in XSQ sibling nodes are aware of each other (by having explicit transitions like $7 => $9), in TurboXPath they are *independent* and only the parent node knows the dependencies and analyzes them on parent element close (lines 20-21 in endElement()). The tree structure helps this operation a lot, as it contains the necessary parent-child dependency information.

This implies that for predicate queries:

1. FAs are more complicated to build, since they require the sibling interdependencies analysis.
2. FAs will occupy more memory as opposed to O(n) for TurboXPath, where n is number of PT nodes. The memory requirement implication is less important as the number of nodes in PT is usually insignificant comparing to the number of elements in the XML stream. In the example above, PT had 5 nodes and XPDT had 9 states.

Indeed, it does look like XPDT (FA equipped with predicates on transition edges) has the same expressive power as the tree model extended with the matching array (note that the example we showed does not illustrate conditional transitions, but they are part of the XPDT model).

However, predicate support is more difficult in FA-based algorithms as the predicate evaluation does not fit neatly into the main model (unlike in trees). The implementation of this support is complex, this is why we usually don't see an extensive predicate coverage in FA algorithm implementations.

To answer one of the block 2 questions, we do not classify any of the XPathMark queries as impossible to implement with FA. It is just the correct implementation is very difficult, so they usually would not be implemented (or get implemented incorrectly, just like in XSQ).

Lastly, an interesting idea would be to merge the two models – i.e. extend an FA algorithm with the TurboXPath-style matching array. It is workable if we have an FA extended with predicates on transition edges (e.g. XPDT), since in this case we can use matching array lookups in the transition predicate expressions.

5.2. Performance

All our performance analyses were taken on elaine2, 900 MHz 2-CPU Solaris 2.8 machine with 2 GB of memory. Unless noted otherwise, we used the XPathMark original document (60K text, 785 elements, depth of 11) containing Internet auction style data.

We will first show the results using regular XPathMark benchmark (uses fixed document size/depth and query types) and later evaluate the performance again using variable document size/depth and/or various query types.

We consider our benchmark results more accurate comparing to these provided by [9] as:

1. We compared two Java implementations (and not C++ with Java implementations).
2. Our implementations were using the same SAX parser ([6]).

5.2.1. Speed

We ran all queries that XSQ processes correctly (Q2, Q3, Q4, Q22) on both XSQ and xstream, averaging over the query time and calculating the QPS.

The part measured was evaluation only (i.e. XPath parsing time was excluded). This is because in most of the scenarios, the queries are fixed and the documents are not, so we are more interested in the XPath evaluation speed as opposed to the XPath parsing speed.

Our observation is that xstream consistently outperforms XSQ, on a rate of 25 to 60 percent. On this particular set of queries, we measured xstream QPS as 5.75, as opposed to 4.39 for XSQ. I.e. xstream is 30% faster.

Then, we turned our attention to the *relative overhead*: how much does xstream add on top of just plain SAX processing (which is a must for both algorithms)? In order to determine this, we wrote xdummy – a simple SAX processor that parses the XML document and does nothing.

Based on the xstream measurements, 81% of the time is spent on XML parsing and only 19% -- on the XPath evaluation. Therefore, TurboXPath does not add a lot of extra cost on top of the plain SAX parsing.

5.2.2. Memory usage

We had not planned to provide detailed memory usage analysis, but we observed an interesting trend in the following maximum memory sizes during the experiments above:

Implementation	Maximum memory usage, MB
xdummy	53
xstream	53
XSQ	58

I.e. xstream adds virtually nothing on top of the plain SAX parsing (of course, this is not exactly true since we do allocate sizable chunks of memory). Nevertheless, while xstream relative memory use was less than 1 MB (not visible in 'top'), XSQ adds a very well measurable 10% on top of the plain processing.

5.2.3. Document size

We took the same queries from the previous section and ran them on the same document copy-pasted 10 times and 100 times. Here is the QPS data we measured:

Size	XSQ	xstream
60K	4.39	5.75
600K	0.87	1.91
6M	0.10	0.53

xstream is a clear winner, more than twice faster as XSQ on 600K document and 5 times faster on the 6M input.

Also, note that both the implementations kept their memory usage constant comparing to the original 60K document processing.

5.2.4. Document depth

One experiment we conducted was to see the effect the recursion depth has on query evaluation. We created documents of the following template:

<a>…<a><b/></a>…</a>

where <a> and </a> tags are present variable number of times.

We tried to issue the query //a/b. This query must keep matching all open a tags until they close, since while the a tag is open, it can always be followed by another b (which, in this case, would be output as a result).

The below table shows the QPS for different depths:

Depth	XSQ	xstream
10	18.11	15.82
50	9.52	7.43
100	n/a	5.81
200	n/a	4.06
400	n/a	3.11
800	n/a	2.80

Unfortunately, the XSQ numbers were not completed as it hangs when the depth is 63 or more.

The results are consistent with the earlier statement [4] that the TurboXPath behavior is linear in terms of the recursion depth.

The memory usage was mostly unaffected for lower depths and grew slightly (+1MB) for higher depths.

Lastly, an interesting fact is that while QPS of xstream for 800 nested nodes is 2.80, the QPS of xstream for 800 sequential nodes (i.e. the same file size/element count) is 7.40.

5.2.5. Query types

There are only three features of XPath that can cause queries to run slower. They are descendant axis, 'any' node test, and predicates.

5.2.4.1. Descendant axis

When processing a location step with descendant axis, if the node test is satisfied, the location step will still be matched in the future in case the recursion happens. Therefore, the number of location steps to match grows by one in this case.

We already analyzed the performance of descendant axis in the recursive case (see previous section).

To estimate the behavior in non-recursive case, we took Q2, /site/closed_auctions/closed_auction/annotation/description/parlist/listitem/text/keyword, as our child axis query and //site//closed_auctions//closed_auction//annotation//description//parlist//listitem//text//keyword as our descendant axis query. We ran the queries on our regular XPathMark document. Since the document is non-recursive in terms of the query, both queries return the same results.

We obtained the following QPS numbers:

/, XSQ	//, XSQ	/, xstream	//, xstream
5.00	4.38	5.81	5.64

For xstream, the performance of the axes is virtually identical. However, descendant axes are noticeably slower for XSQ.

The generic difference between XSQ and xstream performance is in line with what we've experienced earlier.

5.2.4.2. 'Any' node test

Any node test is expressed in XPath as '*'. For example, querying for any element that has sub-element price greater than $50, is //*[price > 50].

'Any' node test introduces uncertainty, just like the descendant axis, and can severely affect the performance, especially when both are used together.

XSQ does not support 'any' node test, so we were not able to compare the implementations. For xstream, we experimentally confirmed that 'any' node test performance is linear in the number of nodes it matches.

5.2.4.3. Predicate evaluation

Predicate evaluation can incur performance overhead for two reasons:

1. Time needed to evaluate the predicate.
2. Since we can see the output node before the predicate evaluates to true, we might need to buffer the output. For predicate-free queries, the matching elements can be output as they are streamed. Buffering effectively makes one extra pass over the matching data and will, therefore, take more time.

To test the predicate performance, we created the following document:

<a><c>11</c> … <c>11</c><c>10</c></a>

where there are 10,000 c elements with the value of 11 followed by one with the value of 10.

We compared two queries. First, //a, is predicate-free. Second, //a[c < 11], is a predicate query. Both queries will return the same result (the entire document).

Also, note that only the last c element satisfies the predicate, so buffering of the entire a element is mandatory. In addition, 10,000 predicate evaluations will be performed.

Here are the QPS results:

Query	**XSQ**	**xstream**
//a	0.2	3.27
//a[c < 11]	0.17	2.90

Here we again experienced the fact the XSQ does not scale very well. With that said, the relative cost of predicate processing is consistent – approximately 10% for both the implementations.

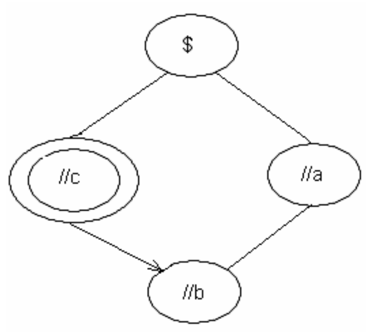
6. FUTURE DIRECTIONS

We believe that xstream provides a stable starting point for a fully functional XPath stream evaluator. Given the project time constraints, it was not feasible to implement the entire language. Therefore, the future directions would be to build on the current implementation and cover a bigger XPath subset. Some of the things that can be added are described below.

6.1. Backward axis processing

This can be done by adding the Xaos query transformation [10] as part of the PT post-processing. We only provided partial implementation of the transformation, handling only the cases where the transformation preserves the PT tree properties.

In certain cases, however, the transformation will break the tree properties of the PT and will turn it into a DAG. For example, applying Xaos on the query //a//b/ancestor::c will turn it into and this is something TurboXPath (yet?) does not know how to handle.



6.2. Function support

XPath language contains several predefined functions. Also, new functions can be defined.

We are aware of no prior attempt to implement XPath functions on a stream. However, we suggest representing the function calls as function trees, in a manner identical to the predicate trees. Then the function evaluation can just reuse the same predicate evaluation model.

We successfully implemented this idea for the contains() and count() functions. More functions can be supported using the suggested model.

6.3. *Expression types*

XPath is a rich language allowing many different types of expressions, including arithmetic expressions, boolean expressions, and so on.

In this project, we only implemented expression types covered by our benchmark. Many more can be implemented on top of the already created infrastructure.

6.4. *Predicate pipelining*

In addition, no research so far has addressed the XPath feature of being able to have multiple predicates off the same node. In this case, the predicates should be applied in the "pipeline" fashion, i.e. the input of predicate i is the output of predicate i-1.

For example, for the following document:

```
<bib>
    <book>
        Unix
        <price>29.99</price>
    </book>
    <book>
        Windows
        <price>39.99</price>
    </book>
</bib>
```

//book[2][price > 35] should match the Windows book, whereas //book[price > 35][2] should result in no matches.

Note that the positional predicates are 1-based in XPath.

This might be an area of a limited practical benefit, but we see it as very challenging to solve from a theoretical standpoint. So far no known algorithm (including TurboXPath and XSQ) can handle multiple sequential predicates.

7. CONCLUSION

In this report, we presented xstream, our implementation of the TurboXPath streaming algorithm. The implementation task was not straightforward. We had to fix several bugs in the algorithm pseudo-code, extend and enhance it to perform the XPath evaluation. We ran XPathMark benchmarks on our implementation, comparing it with the XSQ both functionality-wise and performance-wise.

From the functionality standpoint, the XSQ covers a disappointing 17%. This was a surprise to us, considering the fact that it is the most functional of the existing FA algorithms. However, in an afterthought it makes sense, given the fact that 73% of benchmark queries contain predicates, which require complex processing (inter-sibling analyses) in the FA model. Despite the above, we consider XPathMark a good benchmark, since this is a type of queries a real application would generate. Our implementation was able to achieve twice as better functional coverage accordingly to the benchmark.

From the performance perspective, both implementations performed reasonably well, though xstream has a steady upper hand (30% faster). In any case, the 81% of the evaluation time is spent just parsing the XML, so the relative overhead of both the implementations is acceptable.

We believe that our benchmarks provide a more precise picture of the performance difference comparing to the prior work [9], since here we use the same runtime environment (JVM) and the same XML parser (Xerces) for both the implementations.

In general, it looks like the major advantage of TurboXPath over XSQ is its usage of the matching arrays. Since we could potentially extend XSQ with these arrays, it is probably too early to discredit the entire FA enterprise.

Streaming XPath evaluation remains a significant technical challenge, due to both the richness of the language and the limitation of having only a single pass on the data. While our implementation provides a successful first step, more areas of the XPath language are still to be addressed, including function support and predicate pipelining.

8. REFERENCES

[1] James Clark and Steve DeRose. XML Path Language (XPath)http://www.w3.org/TR/xpath
[2] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language http://www.w3.org/TR/xquery/
[3] James Clark. XSL Transformations (XSLT) http://www.w3.org/TR/xslt
[4] Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. On the Memory Requirements of XPath Evaluation over XML Streams http://www.almaden.ibm.com/cs/people/fontoura/papers/pods2004.pdf

[5] Feng Peng and Sudarshan S. Chawathe. XSQ A Streaming XPath Engine (Technical Report) http://www.cs.umd.edu/projects/xsq/xsqtr.ps
[6] Xerces2 Java Parser 2.6.2 Release http://xml.apache.org/xerces2-j/
[7] Massimo Franceschet. XPathMark*:* An XPath benchmark for XMark http://staff.science.uva.nl/~francesc/xpathmark/benchmark.pdf
[8] XMark — An XML Benchmark Project http://monetdb.cwi.nl/xml/index.html
[9] Vanja Josifovski, Marcus Fontoura, and Attila Barta. Querying XML Streams http://www.almaden.ibm.com/cs/people/fontoura/papers/turboxpath.pdf
[10] Charles Barton, Philippe Charles, Deepak Goyal, Mukund Raghavachari, Marcus Fontoura, and Vanja Josifovski. Streaming XPath Processing with Forward and Backward Axes http://www.cs.nyu.edu/~deepak/publications/icde.pdf
[11] Mehmet Altınel and Michael J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination http://www.cs.berkeley.edu/~franklin/Papers/XFilterVLDB00.pdf
[12] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering http://www.cs.berkeley.edu/~diaoyl/publications/yfilter-tods-2003.pdf
[13] Byron Choi. What are Real DTDs Like? http://www.cis.upenn.edu/~kkchoi/realdtds.pdf
[14] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML Streams with Deterministic Automata http://www.cs.washington.edu/homes/suciu/paper-techreport.pdf
[15] Yanlei Diao, Mehmet Altinel, and Michael J. Franklin. NFA-based Filtering for Efficient and Scalable XML Routing http://sunsite.berkeley.edu/Dienst/Repository/2.0/Body/ncstrl.ucb/CSD-01-1159/pdf

9. APPENDIX 1: DEMO

All our work can be reviewed at **/afs/ir.stanford.edu/users/o/l/olegs/public/cs276b/demo**. The bin sub-directory contains all executables, the src directory – the source code.

9.1. Usage

xstream <file> <query>

where

file – the XML file to evaluate query against
query – the XPath query

xstream will print out only sections of <file> that match the <query>. The last output line is the evaluation time (in milliseconds).

For example:

```
elaine1:~/public/cs276b/demo/bin>            ./xstream
document1.xml "//keyword/bold"
<bold> giving stood stagger </bold><bold> cell
rivers flesh loyal pith </bold>
155
```

9.2. Files

9.2.1. bin

1. Scripts.
   perf – performance measurement script.
   xsq – XSQ helper script.
   xstream – xstream helper script.

2. Java libraries.
   grappa1_2.jar – XSQ GUI package (needed for it to run).
   xercesImpl.jar – Xerces parser.
   xsqf.jar – XSQ package.
   xstream.jar – xstream package.

3. XML files.
   document1.xml – XPathMark benchmark file.
   document1_xsq.xml – XPathMark benchmark file, slightly modified for XSQ needs.
   P9-benchmark.xml – XPathMark benchmark description, including correct answers.

4. Query files.
   perf.in – performance benchmark queries.
   xmark.in – all XPathMark queries.

9.2.2. src

QueryTree.java – PT class.
QueryTreeBuilder.java – XPath parser invocation, PT construction.
TurboXPath.java – implementation of the evaluator algorithm.
makefile – make file to build the source.
xstream.java – the main program.
xdummy.java – the plain SAX XML parsing program (no evaluation).

10. APPENDIX 2: XPATHMARK QUERIES

Q1 /site/regions/*/item
Q2 /site/closed_auctions/closed_auction/annotation/description/parlist/listitem/text/keyword
Q3 //keyword
Q4 //listitem//keyword
Q5 /site/regions/*/item[parent::namerica or parent::samerica]
Q6 //keyword/ancestor::listitem
Q7 /site/open_auctions/open_auction[bidder[personref/@person='person0']/following-sibling::bidder[personref/@person='person1']]
Q8 /site/open_auctions/open_auction[@id='open_auction0']/bidder/preceding-sibling::bidder
Q9 /site/regions/*/item[@id='item0']/following::item
Q10 /site/open_auctions/open_auction/bidder[personref/@person='person1']/preceding::bidder[personref/@person='person0']
Q11 id('person0')/name
Q12 id(/site/people/person[@id='person1']/watches/watch/@open_auction)
Q13 id(id(/site/people/person[@id='person1']/watches/watch/@open_auction)/seller/@person)
Q14 id(/site/closed_auctions/closed_auction[buyer/@person='person4']/itemref/@item)[parent::europe]
Q15 id(/site/closed_auctions/closed_auction[id(seller/@person)/name='Kaivalya Potorti']/itemref/@item)
Q16 /site/people/person[address and (phone or homepage)]
Q17 /site/people/person[not(id(watches/watch/@open_auction)/reserve)]
Q18 /site/open_auctions/open_auction/bidder[position()=1 and position()=last()]
Q19 /site/open_auctions/open_auction[count(bidder)>5]
Q20 /site/regions/australia/item[location='United States']
Q21 /site/regions/*/item[contains(description,'gold')]
Q22 /site/closed_auctions/closed_auction[price<50]
Q23 /site/closed_auctions/closed_auction[id(seller/@person)/profile/@income > 2 * id(buyer/@person)/profile/@income]