Benchmarking Stream-based XPaths: Supporting Simultaneous Queries for Service-Oriented Networking

T. C. Lam, Stanley Poon, and Jianxun Jason Ding

Abstract—Stream-based simultaneous XPaths play a critical role in service-oriented networking, where the processing must scale well in terms of concurrent input streams and number of XPath queries. However, there are no benchmarks or evaluation methodologies in existing literatures that benchmark stream-based XPaths supporting simultaneous queries. In this paper, we describe a novel benchmarking methodology for evaluating XPaths which handle simultaneous queries on streaming traffics. With structured data model, query model, and control model in our benchmark, we conduct well controlled experiments to assess and isolate various performance factors. We also demonstrate that our structured, quantified approach with wide data set coverage, enables accurate performance measurements, and easy bottleneck isolations of a real-world XPath engine implementation.

Index Terms—Benchmark, XML, XPath, Streaming

I. INTRODUCTION

Extensible Markup Language (XML) is broadly used in database and networking. It is the de facto standard for the interoperable data format. Thus, the performance study of XML query engines is a critical issue [1].

XPath is a standard query language for XML [2]. In Cisco's Application Oriented Network (AON) [3], numerous XPaths in an intelligent network device are simultaneously evaluated with streams of XML documents. Stream-based simultaneous XPath processing requires timely response in matching huge volume of input data. For example, a retailer’s delivery system based on RFID may process millions of small files, such as events of a few hundred bytes, and route the events from the scanners at the warehouse to the enterprise network with low latency. One the contrary, a financial clearinghouse may process a few hundred megabytes of consolidated historical records and pass them onto reporting and analytics applications running in the data center.

Due to the large variety of XML traffics and XPath queries in the networking environment, we need to characterize the XPath engines with multiple metrics that allow objective comparisons under different applications’ needs. In this paper, we propose a benchmarking methodology for the stream-based simultaneous XPath engines. We focus on carefully quantified data model, query model, and control model. The data model captures the natures of XML streams. The query model captures the natures of XPath queries. The control model captures various execution environment settings. Our structured quantitative design allows accurate measurements for one factor at a time. In contrast, existing benchmarks prepare data sets from real traffic traces or arbitrarily generate them from XML generators. They measure a few performance factors, such as document size, and usually do not specify their exact ranges of values. As a result, they cannot isolate individual performance factors and are less effective for designing and evaluating of real-world XPath engines.

The rest of this paper is organized as follows. We describe the related work in Section II. Then we explain our data, query, and control models in Sections III, IV, and V. We use the results obtained from a real-world XPath engine to show the superiority of our benchmarking approach in Section VI, and conclude our paper in Section VII.

II. RELATED WORK

There are numerous existing benchmarks designed for XML query engines, such as NEXMark [4], XPathMark [5], XBench [6], XMark [7], Michigan Benchmark [8], XOO7 [9], XMach-1 [10]. However, they are not suitable for streaming. First, they evaluate one query/document at a time, whereas for streaming, we need to assess the scalability with different number of XML streams, simultaneous queries, and computing resources. Second, except NEXMark, they are designed for the DOM based parsing model [11]. DOM is widely used for database applications, but is not suitable for streaming due to its large memory consumptions. Third, except XPathMark, they are not designed for XPath. The goal of our benchmark is to provide accurate benchmarking and problem isolation capabilities for implementers of stream-based XPaths.

III. DATA MODEL

The natures of input data (XML document) being processed can influence the query performance. Thus, we need to design the data model in such a way that it can capture the performance impacts caused by individual characteristics of the documents. A list of characteristics being studied is summarized in Table I.

Many existing benchmarks also study similar characteristics, but their data sets are not suitable for controlled experiment. For example, processing a 5Kbyte document could be 2x faster than

Manuscript received July 22, 2008.

T. C. Lam is with Cisco System, Inc., San Jose, CA 95134 USA (e-mail: brlam@cisco.com).
Stanley Poon is with Cisco System, Inc., San Jose, CA 95134 USA (e-mail: spoon@cisco.com).
Jianxun Jason Ding is with Cisco System, Inc., San Jose, CA 95134 USA (e-mail: jding@cisco.com).

978-1-4244-2324-8/08/$25.00 © 2008 IEEE.

This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE "GLOBECOM" 2008 proceedings.
processing a 10Kbyte document. But if the two documents have
their other properties, such as attribute/node density, extremely
different, then we cannot conclude that the 2x speedup is solely
contributed by the change in document size. Another advantage
of our benchmarking methodology is that we can extend the data
sets easily with wide range of data points. As depicted in Figures
1 and 2, our benchmark provides a more comprehensive set of
XML data comparing with other XML data sources, xCBL [12]
and XMLBench [13].

In order to have more controlled characteristics on the data
sets, we design a base document that allows other documents
with various natures to be derived from it. The base document
has balanced binary tree structure, with fixed number of child
nodes per non-leaf node, fixed number of text characters per
leaf node, fixed number of attributes per element, fixed length
of element name, attribute name, and attribute value. In the base
document, no namespace is used. The base document and the
documents derived from it can be generated by XML generator
such as ToXgene [14] with minimum handcraft efforts.

A. Document Depth and Size

To double the depth of a document, we first create a dummy
child at the rightmost leaf-node of the root’s left sub-tree. Then,
we move the root’s right sub-tree to be the child of this dummy
node as depicted in Fig. 3. However, it will decrease the number
of leaf-node by one, and hence, will decrease the number of text
characters. To maintain the number of text characters, we create
another dummy child at the root that has roughly the same text
length as other leaf-node. The total number of nodes increases
by two, and hence, increases the node density, but the impact is
insignificant. To maintain tag density and attribute density, we
assign short element names and no attributes for dummy nodes.

To double the size, we first duplicate all sub-trees of the root,
and then assign a dummy node between them, as depicted in Fig.
3. The number of nodes is doubled, but the number of leaf node
is one more than double. By not assigning any text characters to
this dummy leaf-node, the impacts to the tag density, attribute
density, and node density are insignificant.

B. Tag Density, Attribute Density, and Node Density

To double the tag density we first increase the length of each
element name, while remaining the lengths of attribute names
and attribute values unchanged. Then we reduce the number of
text characters accordingly until the document size is about the
same as the original document. For example,

\[
<\text{element_name} \ attr_n="\text{attr_v}" >12345\text{<element_name>}
\]

is converted to

\[
<\text{element_name L1 attribute_name="attribute_value" } >1234
\\text{<element_name L1>}
\]

to double the attribute density (1), we need a document with
high tag density (long element names) derived in the previous
step. We first increase the length of attribute name and value.
Then, we reduce the length of element name until the tag length
is about the same as the original document. For example,

\[
<\text{element_name at level L1 attr_n="attr_v" } >1234
\\text{<element_name at level L1>}
\]

is converted to

\[
<\text{element_name L1 attribute_name="attribute_value" } >1234
\\text{<element_name L1>}
\]

to double the attribute density (2), we need a document with
high attribute density (1) (long attribute names) derived in the
previous step. We first break down each long attribute into two
shorter ones. The total length of the attribute is identical to the
sum of those of the two short attributes. For example,

\[
<\text{element_name L1 attribute_name="attribute_value" } >1234
\\text{<element_name L1>}
\]

is converted to

\[
<\text{element_name L1 attr_n0="attr_0" attr_n1="attr_1" } >1234
\text{<element_name L1>}
\]

to double the node density, we need a document with long

\[
\begin{array}{|c|}
\hline
\text{Metrics} & \text{Remarks} \\
\hline
\text{Document Depth} & \text{Nested depth of the document} \\
\text{Document Size} & \text{No. of bytes in the document} \\
\text{Tag Density} & \text{No. of bytes in element tags to document size} \\
\text{Attribute Density (1)} & \text{No. of bytes in attributes to document size} \\
\text{Attribute Density (2)} & \text{No. of attributes to no. of elements} \\
\text{Node Density} & \text{No. of nodes (tag-pairs) to document size} \\
\text{Namespaces} & \text{No. of namespaces used in the document} \\
\hline
\end{array}
\]
element names and attributes derived in the previous step. We first create a dummy sibling for each non-root node, which has no child or text as shown in Fig. 4. Then, we adjust the length of the attribute name and value for each non-root node, including the dummy sibling, to be half of that in the original document. Then, we decrease the length of the element name until the total tag length is around half of the original document. For example, 

\[
<\text{element}_n \text{attribute}_n=\text{"attribute_value"}> \\
<\text{a}>xxx</\text{a}></\text{element}_n>
\]

is converted to

\[
<\text{elm}_n \text{attr}_n=\text{"attr_v"}>>\text{a}>xxx</\text{a}></\text{elm}_n>
\]

\[
<\text{elm}_n \text{attr}_n=\text{"attr_v"}>>></\text{elm}_n>
\]

IV. QUERY MODEL

Given the data sets generated in the previous section, we can design the corresponding XPath to assess various query natures. A list of characteristics being studied is summarized in Table II. We group the characteristics into three categories. The first one focuses on the size and similarity of a collection of simultaneous XPaths. The second one focuses on the complexity of particular XPath operations. The third one focuses on the selectivity of the XPath against the input XML document.

Unless otherwise specified, each XPath in the query sets has a unique matched result closed to the end of the XML document. The XPaths are designed as simple as possible to avoid any side effects from influencing the particular performance factor being measured. Table II shows simplified examples of the query sets. The number next to the XPath expressions in the table indicates the quantitative measure of the corresponding metric. We will explain some of them if they are not self-explaining in the table.

A. XPath Collection

While most existing work focuses on one XPath at a time, we consider processing of a collection of XPaths simultaneously to evaluate the scalability of XPath engines and the performance impact due to the correlations between XPaths. For scalability tests, we vary the number of simple XPaths. A simple XPath has no wildcard operator (* or //), no predicates, and no extractions, with fixed lengths from 5 to 10 location steps. More complex XPaths are evaluated in other test cases. For correlation tests, we vary the degree of similarity between XPaths by the number of common steps. For example, in Table II, Set A shares /a1 in both XPaths so the number next to them is 1. Set B shares /a1/b1 so the number is 2. Set C shares /a1/b1 so the number is 3. This can measure how effective implementations of XPath engines in leveraging similarity between XPaths to improve performance.

B. XPath Complexity

While most existing benchmarks focus on extensive study of XQuery features, our benchmark focuses on the characteristics of particular XPath operations. As shown in Table II we vary the length of XPath, the number of wildcard operators (* and //), the number of predicates, and the complexity of predicates.

To measure the impacts due to the number of predicates, we use simple predicate which contains an attribute in the form of [/@x=’1’] for an element. To measure the predicate complexity we design the XPaths in such a way that can measure the costs of different predicate types incrementally. For example we can use XPaths below to isolate the costs of [@x, text(), and substring()]:

- \( \text{boolean}(\text{a1}/[@x=’1’]) \)
- \( \text{boolean}(\text{a1}/[@x=’1’ \text{ and } \text{text}()<’12’]) \)
- \( \text{boolean}(\text{a1}/[@x=’1’ \text{ and } \text{substring(text(),1,2)<’12’}]) \)

C. XPath Selectivity

The selectivity of an XPath measures the percentage of nodes that are matched by the XPath. The cost of extracting the result nodes is a significant portion of the overall evaluation. Thus, it

<table>
<thead>
<tr>
<th>Categories</th>
<th>Metrics</th>
<th>Set A</th>
<th>Set B</th>
<th>Set C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection</td>
<td>No. of queries</td>
<td>1: boolean(a1/b1/c1/d1/e1)</td>
<td>2: boolean(a1/b1/c1/d1/e1)</td>
<td>3: boolean(a1/b1/c1/d1/e1)</td>
</tr>
<tr>
<td></td>
<td>XPath sharing</td>
<td>1: boolean(a1/b1/c1/d1/e1)</td>
<td>2: boolean(a1/b1/c1/d1/e1)</td>
<td>3: boolean(a1/b1/c1/d1/e1)</td>
</tr>
<tr>
<td></td>
<td>XPath length</td>
<td>3: boolean(a1/b1/c1/d1/e1)</td>
<td>4: boolean(a1/b1/c1/d1/e1)</td>
<td>5: boolean(a1/b1/c1/d1/e1)</td>
</tr>
<tr>
<td></td>
<td>* density</td>
<td>1: boolean(*/b1/d1/e1)</td>
<td>2: boolean(*/b1/d1/e1)</td>
<td>3: boolean(*/b1/d1/e1)</td>
</tr>
<tr>
<td></td>
<td>// density</td>
<td>1: boolean(a1/b1/c1/d1/e1)</td>
<td>2: boolean(a1/b1/c1/d1/e1)</td>
<td>3: boolean(a1/b1/c1/d1/e1)</td>
</tr>
<tr>
<td></td>
<td><em>/</em> // vs. //</td>
<td>1: boolean(<em>/</em>/c1/d1/e1)</td>
<td>2: boolean(<em>/</em>/c1/d1/e1)</td>
<td>3: boolean(<em>/</em>/c1/d1/e1)</td>
</tr>
<tr>
<td></td>
<td>No. of predicates</td>
<td>1: boolean(a1[@x=’1’])</td>
<td>2: boolean(a1[@x=’1’])</td>
<td>3: boolean(a1[@x=’1’])</td>
</tr>
<tr>
<td></td>
<td>Predicate complexity</td>
<td>3: boolean(a1[@x=’1’])</td>
<td>3: boolean(a1[@x=’1’])</td>
<td>3: boolean(a1[@x=’1’])</td>
</tr>
<tr>
<td>Selectivity</td>
<td>Extraction</td>
<td>0: boolean(a1/b1/c1/d1/e1)</td>
<td>5: boolean(a1/b1/c1/d1/e1)</td>
<td>10: boolean(a1/b1/c1/d1/e1)</td>
</tr>
<tr>
<td></td>
<td>Mismatch</td>
<td>1: boolean(a1/x/b1/c1/d1)</td>
<td>2: boolean(a1/x/c1/d1/e1)</td>
<td>3: boolean(a1/x/d1/e1)</td>
</tr>
</tbody>
</table>
is critical to study how selectivity affects the performance.

For extraction tests, we derive numerous XPaths from a base XPath, e.g., /a//b/c//d/e, which has a unique match result with a deepest level leaf-node close to the end of the XML document. To achieve zero-extraction rate we enclose the base XPath with a Boolean operator as shown in Set A of Table II. To extract the text characters of one leaf node (e_i) we append a text() operator at the end of the base XPath, as shown in Set B of Table II. To double the extraction rate, we do not modify the XPath. Instead, we modify the element names of other nodes in the document so that the XPath has matches with two paths of identical element names in the document. Therefore, as shown in Sets B and C in Table II, they have identical XPath but their extraction rates are different (5% and 10%).

For mismatch tests, we modify the Boolean form of the base XPath in such a way that it will find matches at different steps. For example in Set A of Table II, we substitute a_i by x so that it will find a mismatch immediately at step 1. In Set B, we substitute b_i by x so that it will find a mismatch at step 2. Similarly, in Set C, we substitute c_i by x so that it will find a mismatch at step 3.

V. CONTROL MODEL

Besides the XML data sets and XPath query sets, it is also crucial to assess optimum concurrency settings for stream-based simultaneous XPath engines, as summarized in Table III.

Different from database applications whose documents have been entirely stored in the server, stream-based XPath engines receive documents packet by packet from the input streams. The pattern of incoming traffics and the size of packets can affect performance due to different context switching capabilities. To distinguish the context switching capabilities of XPath engines, we investigate three traffic patterns: in-order, round-robin, and random, as depicted in Figure 5. The in-order pattern processes packets in a session one by one before processing the packets in another session. It requires the least context switching among the three patterns. The round-robin pattern processes the first packet in each session, then the second packet in each session and then the third one, etc. It requires the most context switching among the three patterns. The random pattern arbitrary selects a session and process the next packet in the session. It requires the medium amount of context switching among the three patterns.

Besides the characteristics of incoming traffics, we also study other concurrency factors, such as the number of CPU cores, the number of sessions, the number of threads used per CPU core, and the number of sessions assigned per thread.

<table>
<thead>
<tr>
<th>TABLE III</th>
<th>PERFORMANCE FACTORS STUDIED IN CONTROL MODEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metrics</td>
<td>Remarks</td>
</tr>
<tr>
<td>Traffic Patterns</td>
<td>In-order, round-robin, or random</td>
</tr>
<tr>
<td>Packet size</td>
<td>No. of bytes per packet</td>
</tr>
<tr>
<td>CPU cores</td>
<td>No. of CPU cores used</td>
</tr>
<tr>
<td>Sessions</td>
<td>No. of concurrent sessions</td>
</tr>
<tr>
<td>Threads-core ratio</td>
<td>No. of threads per CPU core</td>
</tr>
<tr>
<td>Session-thread ratio</td>
<td>No. of threads per session</td>
</tr>
</tbody>
</table>

VI. BENCHMARK RESULTS

In this section, we will demonstrate how our benchmark is used in evaluating a real-world stream-based XPath engine with support for simultaneous XPath. Our benchmark contains over 400 test cases across more than 18 performance factors. We measure throughput, latency per packet, and memory usage in each test case. We execute sufficient repetitions of experiments until the measurement variants are within 1% for single CPU, and 2% for quad CPU. For XPath engines which have separate parsing and XPath query, we measure two sets of data, one for parse-only and the other for both parsing and XPath query. As shown in Figure 6, it allows us to isolate performance cost among the two layers and make optimization decisions.

To illustrate the merits of the benchmark, following sections present results and analysis obtained from a stream-based XPath engine developed at Cisco Systems, Inc, which implementation details will be described separately in future publications. In this paper, we limit our focus on the benchmarking methodology.

A. Test Platform and Settings

The hardware and software configurations of our benchmark are summarized in Table IV. In our control experiments, each test case measures the impact of one performance factor. The targeted factors will take values as specified in the test cases. Other factors will take default values as stated in Table V.

Fig. 5. Varying the traffic patterns of input XML streams.

Fig. 6. Parsing overhead.
B. Data Complexity

Figure 7 shows how performance varies with document sizes. The XPath engine scales well with large documents. For small documents, bigger latency is due to initialization cost. For larger documents, this static overhead is amortized to approach zero. Therefore, comparing the data for 1Kbytes and 100Mbytes, we can deduce that the static overhead is 22μs.

Figure 8 shows the performance impacts of node density and attribute density. We scale their density values by 1/5K and 10x, respectively, for convenience of display. When the node density increases from 10 to 200 in the figure, the latency is increased by 2x. When the attribute density increases from 32 to 260 in the figure, the latency is increased by 4x.

C. Query Complexity

Figure 9 shows how the XPath engine scales with the number of simultaneous XPaths. There is degradation as more queries are processed. Compared to a simple sequential execution, it is n-order of magnitude better than simultaneous execution. For example, evaluating 100 XPaths take 32*100μs as compared to 39μs in the Cisco’s engine. For 1000 XPaths, the difference is close to 1000x.

Figure 10 compares the costs for descendant-or-self (//) axis versus the child axis (/). When the number of // axis increases from 0 to 6, the latency is increased by 34%. In contrast, the / axis induces almost no additional cost.

Figure 11 compares the selectivity of XPath. It shows that as the extraction of results increases from 3% to 20%, the latency is increased from 27.5μs to 30.5μs. In other word, performance degrades as there are more matches with the document.

Figure 12 shows how the performance varies with complexity of predicate in XPaths. It measures the efficiency of expression evaluator which is the key to most streaming engines.

D. Concurrency and Scalability

Figure 13 demonstrates the scalability with concurrent XML streams. This measures the cost for switching context between sessions. Beyond 1000 sessions, the cost levels out. At that level, the cost to switch context reaches a maximum and any increase in concurrency has no apparent performance impact.

Figure 14 shows the performance impacts on the ordering of packets. In sequential mode, all the packets for the same session arrive before a new session begins. The engine performs better

TABLE IV

<table>
<thead>
<tr>
<th>Performance Factors</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Core</td>
<td>1 Core</td>
</tr>
<tr>
<td>Concurrency</td>
<td>1 session</td>
</tr>
<tr>
<td>Traffic Pattern</td>
<td>Round-Robin</td>
</tr>
<tr>
<td>Packet Size</td>
<td>1500 Bytes</td>
</tr>
<tr>
<td>Threads-core ratio</td>
<td>1 thread per core</td>
</tr>
<tr>
<td>XPath Properties</td>
<td>1 XPath with 5 location steps. No // axis</td>
</tr>
<tr>
<td>Document Properties</td>
<td>5K, 10 levels deep, 6.4 nodes per 1KB, 1 attribute per node</td>
</tr>
</tbody>
</table>

TABLE V

<table>
<thead>
<tr>
<th>Performance Factors</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Core</td>
<td>1 Core</td>
</tr>
<tr>
<td>Concurrency</td>
<td>1 session</td>
</tr>
<tr>
<td>Traffic Pattern</td>
<td>Round-Robin</td>
</tr>
<tr>
<td>Packet Size</td>
<td>1500 Bytes</td>
</tr>
<tr>
<td>Threads-core ratio</td>
<td>1 thread per core</td>
</tr>
<tr>
<td>XPath Properties</td>
<td>1 XPath with 5 location steps. No // axis</td>
</tr>
<tr>
<td>Document Properties</td>
<td>5K, 10 levels deep, 6.4 nodes per 1KB, 1 attribute per node</td>
</tr>
</tbody>
</table>

Fig. 7. Variation of packet latency with document size.

Fig. 8. Impact of node and attribute density.

Fig. 9. Scalability in terms of simultaneous XPath.

Fig. 10. Cost of the descendant-or-self axis.

Fig. 11. Selectivity of XPath (% of extraction).
because there is fewer context switching. The worst case is the round-robin mode. The random mode comes in between them.

Figure 15 shows the memory usage per session, which is also an important factor for scalability. It shows that memory usage stays flat at 2.3 KB per session for the XPath evaluation. In this figure the memory of parsing is not included because our focus is on the cost of XPath.

Due to limit of space, we only present part of our benchmark results here. Based on these results, the following performance issues are identified: (1) high latency for messages smaller than 1Kbyte; (2) scaling to very large set of XPaths; (3) high parsing overhead and cost of context switching between sessions. These examples demonstrate that our data model, query model, and control model not only benefit us in the XPath engine design, but also help our customers to make proper decisions on sizing their XPath engine server cluster and load balancing incoming traffic to each server node.

VII. CONCLUSION

Stream-based simultaneous XPath engines are in hot demand for enterprise and telecom service provider applications. High-performance is the most important product requirement for such type of engines. However, there is no benchmark or benchmarking methodology published in literatures focusing on the evaluation of XPath engines with capabilities of handling simultaneous queries on streaming traffics. In this paper, we proposed a novel structured modular approach to benchmark stream-based simultaneous XPath engines. We uniquely applied techniques of controlled experiments to systematically measure modular language constructs of XPath. We demonstrated that this new approach allows us to conduct tests on a wide range of quantified test data to isolate performance factors related to the data streams, traffic patterns, application query properties, and implementation choices. As a result, we identified various potential improvements for the engine. All these contribute to building and deploying high-performance stream-based XPath engines supporting simultaneous queries.

REFERENCES