Efficient Evaluation of Multiple Queries on Streaming XML Data

Mong Li Lee, Boon Chin Chua, Wynne Hsu, Kian-Lee Tan
School of Computing, National University of Singapore
3 Science Drive 2, Singapore 117543
{leeml, chuaboo1, whsu, tankl}@comp.nus.edu.sg

ABSTRACT
Traditionally, XML documents are processed at where they are stored. This allows the query processor to exploit pre-computed data structures (e.g., index) to retrieve the desired data efficiently. However, this mode of processing is not suitable for many applications where the documents are frequently updated. In such situations, efficient evaluation of multiple queries over streaming XML documents becomes important. This paper introduces a new operator, mqX-scan, which efficiently evaluates multiple queries with a single pass on streaming XML data. To facilitate matching, mqX-scan utilizes templates containing paths that have been traversed to match regular path expression patterns in a pool of queries. Results of the experiments demonstrate the efficiency and scalability of the mqX-scan operator.

Categories and Subject Descriptors
H.2.4 [Database Management]: Systems – *query processing*.

General Terms
Performance.

Keywords
Streaming XML, Finite State Machines, Regular Path Expressions.

1. INTRODUCTION
The eXtensible Markup Language is widely used as a standard for information exchange on the World Wide Web. This has led database researchers to develop data models and query languages for XML [2, 3, 6, 8, 9, 10]. An XML document is typically modeled as a graph and query languages are based on regular path expressions (RPE) that traverse the logical XML graph model. Current XML query engines and query processing tools assume that XML documents are stored in some local repository. As such, they exploit pre-computed data structures (e.g., indexes) on the XML data to facilitate efficient query evaluations.

In a dynamic environment such as the Internet, the content of XML documents changes continuously and we need to process the XML documents "on the fly". In other words, the XML document is processed as it streams in from the network without prior storage. Furthermore, we frequently find many queries being issued on one XML document. A resourceful query optimizer would try to determine the common expressions in these queries and carry out some group optimization [7]. [11] propose the X-scan operator to process non-materialized XML data. However, this operator parses an XML document once for each query. As a result, if we have *m* queries on the same XML document, then the document will be parsed *m* times. In order to address this problem, we propose a new *mqX-scan* operator to handle *m*ultiple *q*ueries with a *single scan* over the *X*ML data.

The input to mqX-scan is an XML data stream and a set of regular path expression queries. The mqX-scan operator evaluates the traversal paths of the RPE queries and binds the values to the variables when matches are found. Our proposed operator has the following features:

1. A global template containing expressions of the paths traversed is constructed to facilitate matching of RPE queries with the document.
2. ID and IDREFs traversals are handled efficiently by using an IDREF template, ID indexes and a Key-Path (KP) index.
3. Cycles that may be introduced by IDREFs traversals are detected and resolved using a source-destination pair table.
4. Auxiliary structures such as stacks are used to track changes in the states of queries during the scanning of the XML document.

This simple but powerful approach proves efficient in evaluating multiple RPE queries over streaming XML document. In addition, we also demonstrate that the technique developed in mqX-scan is applicable to Selective Dissemination Information system applications which are currently limited to user profiles (queries) that do not involve ID and IDREFs.

The rest of the paper is organized as follows. Section 2 briefly describes how RPE queries are evaluated and Section 3 reviews some related work. Section 4 introduces our new mqX-scan operator and details the evaluation of multiple queries. Section 5 reports the results of experiments to evaluate the performance of mqX-scan. Section 6 concludes our work and discusses some future research direction.

2. PRELIMINARIES
Figure 1 shows a simple XML document consisting of user-defined tags, elements, ID and IDREF references. The corresponding graph representation is given in Figure 2. An element is represented as an edge and a node is labelled with its corresponding ID if any. The IDREFs are shown in the graph as

dashed lines and are represented as edges labelled with the IDREF attribute name. PCDATA is represented as dotted arrows pointing to leaf nodes. Given a query *db/lab/location/_*/city*, we can evaluate it by traversing the graph in Figure 2 starting from the root element "db" to find a match for the first outgoing edge, that is "lab". Then it will follow the outgoing edge "name" from the node "central" to node "#1". Since "name" doesn't match "location", the matching terminates before it matches the entire path expression in the query, so it is not bound to the query. Next, we backtrack to the "central" node and follow the next outgoing edge to node "#2". From node "#2" we traverse the edge "city" and come to the end of the path. The path traversed thus far is db→lab→location→city, which again does not match the path expression in the query. This process is carried out recursively until all the matches to the queries are found.

```
<db>
 <lab ID="central" advisor="Smith1">
  <name>e-commerce lab</name>
  <location>
   <street>Clementi</street>
   <city>Singapore</city>
   <country>Singapore</country>
  </location>
 </lab>
 <lab ID="lab2">
  <name>Security lab</name>
  <location>
   <street>Science Park</street>
   <city>Singapore</city>
  </location>
 </lab>
 <paper ID="Smith2" source="central" researcher="Smith1">
  <title> Wireless....</title>
 </paper>
 <researcher ID="Smith1" nickname="Smith2">
  <lastname>Smith1</lastname>
 </researcher>
</db>
```

Figure 1. A sample XML document.

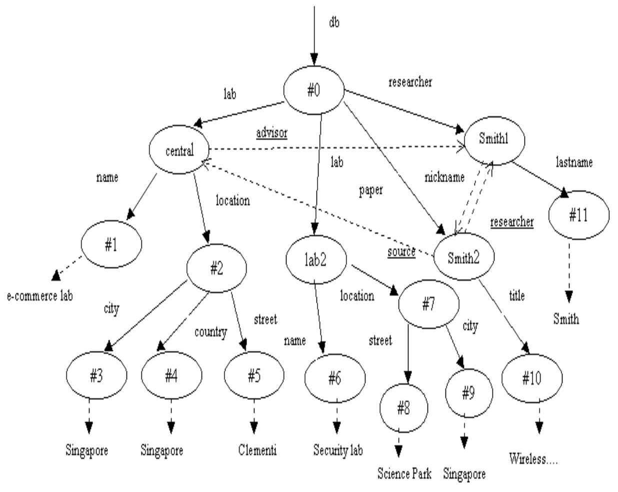

Figure 2. Graph representation for XML document in Figure 1.

When an IDREF is encountered, the traversal will skip to the node that the IDREF points to. For example, the node "central" in Figure 2 has an IDREF attribute advisor = "Smith1". We will then follow the path from the node "central" to the node "Smith1" and traverse down the "Smith1" node to find potential matches. The path for this particular IDREF example would be db→lab("central")→lastname. We observe that each path traversal over the document is the potential answer to a regular path expression query. In Section 4, we will show how the mqX-scan operator makes use of this fact to efficiently process multiple queries on an XML document.

3. RELATED WORK

Streaming data gained prominence with the advent of the Internet and its many applications. An overview of the issues involved in evaluating queries over data stream is given in [5]. With the increasing use of XML as a standard for data exchange and querying, much research has been carried out on the efficient evaluation of regular path expressions [11, 12]. [12] proposes several algorithms for processing regular path expressions. These algorithms deal with searching paths from an element to another, scanning sorted elements and attributes to find element-attribute pairs and finding Kleene-Closure on repeated paths or elements. These methods are applicable in the case where the XML documents are stored in local repositories.

[11] proposes an operator called X-scan to evaluate a set of regular path expressions occurring in a query over an XML data stream. X-scan accepts one query at a time. When m queries (m > 1) are issued on the same document, the performance of X-scan degrades as it will need to parse the same document m times. The mqX-scan operator is designed to address this problem.

[1] develops XFilter, an XML-based system for supporting Selective Dissemination of Information (SDI). In XFilter, user profiles or interests are modeled using Xpath queries. When XML documents stream in, queries are evaluated against a query index that has been pre-built on the queries. If a query matches *at least* one path of the document, then the document is considered to match the query and will be channeled to the user. On the other hand, the mqX-scan operator requires that the queries must match *all* the paths of a document before it can return the bindings to the queries.

4. THE MQX-SCAN OPERATOR

Our proposed new operator, mqX-scan, as its name implies, supports *multiple queries* in a single pass of the XML document. The input to the operator is a pool of regular path expression queries and an XML document. The output of mqX-scan is a set of bindings to the queries. Figure 3 shows the overall architecture and data structures used in mqX-scan.

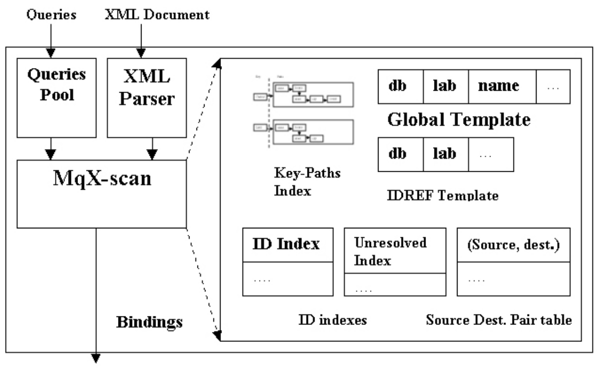

Figure 3. Overview architecture and data structures.

The mqX-scan operator employs several data structures to aid in its query processing:

- A global template that contains the element names encountered on the paths traversed so far. This template facilitates matching of multiple queries in the query pool.
- A Key-Path (KP) index to provide fast ID and IDREFs traversals. The KP index consists of two parts, a key and its corresponding paths.
- An ID index that keeps track of the element IDs that the parser has seen.
- An unresolved ID index that stores yet-to-be-seen IDs. The entries in this index contain suspended IDs in which the KP index has yet to contain the corresponding nodes.
- An IDREF template that contains the paths traversed by IDREFs. This template provides fast matching of queries when the document involves IDREFs.
- A (source, destination) pair table that detects cycles caused by IDREFs traversals.

Figure 4 shows the architecture of the KP index. When we encounter a node with "central" as the ID, we will keep all the sub-elements of the ID node in its corresponding paths. Thus we have the key "central" and the set of paths {<name>}, {<location><street>}, {<location><city>} and {<location><country>}. Note that even though the path name <location> occurs multiple times, the KP index will only store it once. We can easily trace all the sub-elements of the ID node using pattern recognition. For example, in Figure 4, there are two paths from ID node "lab2" to another node: via the edge labeled "name" and the edge labeled "location". This is followed by two paths labeled "street" and "city" emanating from the node the "location" edge points to. This index is clean and small compared to the traditional structural graph index proposed in X-Scan. This is because the KP index only stores the portion of the XML document that involves IDs.

RPE queries will be kept in a query pool for subsequent interaction with the mqX-scan operator during the traversal of the XML document. The operator will match the RPE queries with the path expressions stored in the global template. Any bindings to these queries will be output as they become available. When mqX-scan has finished scanning the whole document, all the bindings to the queries would have been found.

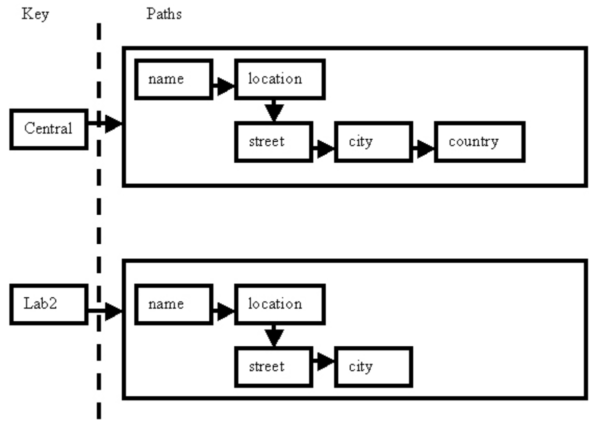

Figure 4. Structure of the Key-Paths (KP) index.

4.1 Matching Queries

Recall the XML document in Figure 2. Conventional approaches to find matches to a query such as *db/lab/_*/city* will traverse a path starting from the "db" edge to an outgoing edge that satisfies the next element, "lab". This is followed by any outgoing edges that will eventually lead to "city". This traversal can be visualized easily by representing the XML document as a graph.

In contrast, the mqX-scan operator uses a global template as a source to match the regular path expressions in the queries. The underlying mechanism of the global template is based on finite state machine (FSM) model. Figure 5 gives the algorithm to find matches for queries. When the parser encounters an element tag, it will call the *startElement* event and pass in the element name, level of the element, and any XML related attribute to the event. A global template will be created to store the element name.

```
Algorithm for evaluation of the queries
Input: An XML document and a set of queries
Output: Bindings to each of the queries

while the document does not end do{
 if parser encounters open tag
        //startElement event handler is invoked
        startElement(...);
 if parser encounters end tag
        //endEvent event handler is invoked
        endElement(...);
 if parser encounters character
        //store the appropriate PCDATA for this traversal
        store the PCDATA

startElement(.., element, ..){
create KP index;
append the element into the template;
....
}
endElement(.., element, ..) {
....
 if the last element in the template matches this element
 //invoke matching function
   for every element in the template
   check if the element matches the expressions in every query
.....
delete the last element in the template
....
}
}// end while
```

Figure 5. Algorithm to evaluate queries.

Consider the graph in Figure 2, when the first element tag <db> is encountered, the *startElement* event is fired, and mqX-scan constructs a global template to store "db". The global template is set to an initial state. Next, the parser reaches "lab" and triggers another *startElement* event. Now the global template contains two elements "db, lab". This process is repeated until the parser encounters an end element tag. It will then invoke the *endElement* event handler to check if the last element in the global template matches the end element tag and set the template to a steady state. Figure 6 shows the state transition diagram for the global template.

When the global template reaches its first steady state, it will invoke the matching function. The matching function matches every element in the global template with every regular path expression in the query pool. In our example, when the global template reaches its first steady state, it will contain "db, lab, name". Queries matching this expression in the template will be bound. Suppose we have two queries, "db/lab/_*" and "db/_*", the matching function will recognize that the template matches these two regular path expressions. Each query in the query pool is associated with a state machine. Initially, all the queries are in a "Wait" state, waiting to be evaluated. The query that is currently being matched is in an "Active" state. The query is placed in a "Bind" state if its regular path expression matches the global template.

Figure 7 shows the State Transition diagram for each query in the query pool. Each query in the queue is evaluated in turn. This round of evaluation ends when all the queries have been examined and are set back to their "Wait" states. Note that when the XML graph is being traversed, only the paths in a "Steady" state template will be matched against the regular path expressions in the queries. When the template is in an intermediate state, no queries will be in an "Active" state.

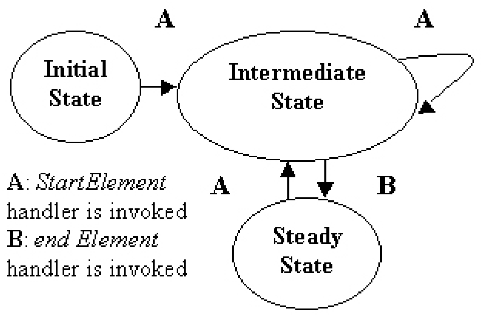
A
A
Initial State
Intermediate State
A: *StartElement* handler is invoked
B: *end Element* handler is invoked
A
B
Steady State

Figure 6. State transition diagram for the global template.

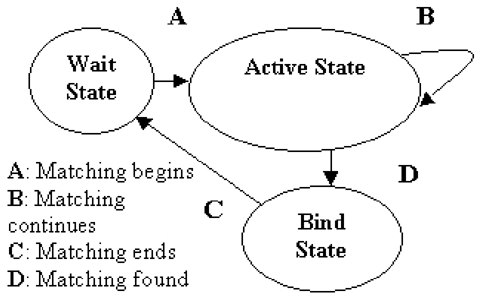

Figure 7. State transition diagram for each query.

4.2 Handling IDREFs and Cycles

We distinguish two types of IDREFs in an XML document: backward references and forward references. As mqX-scan traverses the XML document, it builds a KP index to track the parent-child relationship between the nodes. The index facilitates fast traversal of backward IDREFs. In order to ensure that the overheads needed for the index structure are minimal, we do not store the leaf nodes. In forward references, the IDREF points ahead to a node that has yet to be processed by the operator and hence, will not appear in the KP index. To handle this, we maintain an ID index on the nodes that has already been processed, and a list of unresolved IDREFs that specifies the ID values and addresses of nodes yet to be "seen" by the parser. When a forward reference is encountered, its ID is suspended and added to the unresolved IDREF list. When a *startElement* event is fired, we check whether the ID of this particular element has been previously suspended. If it is, then the suspended ID will be activated. After all the sub-elements of this element have been processed, its ID will be removed from the list of unresolved IDREF and added to the ID index.

In addition, we introduce another template, called the IDREF template, which is similar to the global template except that it has an initial "Inactive" state. Figure 8 depicts the state transition diagram for IDREF template. If the XML document does not contain IDREF, then the ID index will be empty and the IDREF template remains inactive. When some IDREF is encountered, the IDREF template will search for the ID in the ID index. Using this ID as the starting point (key), we will probe every sub-element in the KP index. Once the IDREF template has found the desired paths, it will invoke the matching function. For example, when mqX-scan encounters the "Smith1" IDREF reference, it will look up the ID index for "Smith1". Since "Smith1" has not yet been processed, it will put it into the unresolved ID index. When "Smith1" is reached, the mqX-scan operator will traverse down the "Smith1" node and append "lastname" to "db, lab". The IDREF template will now have "db, lab, lastname". This template is used to match the queries.

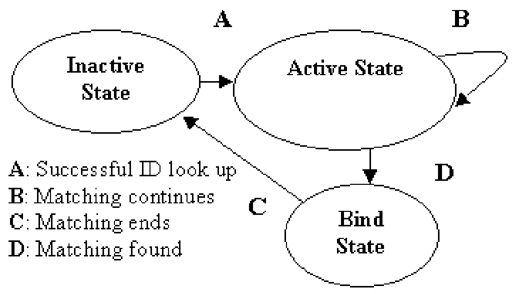

Figure 8. State transition diagram for IDREF template.

Finally, we maintain a (source, destination) pair table to detect and handle cycles. When we encounter the ID "central" and its IDREF pointing to the other ID of "Smith1", we store "central" as a source in the table. When the ID index contains the ID "Smith1", we store this ID as a destination. Thus, we have the pair (central, Smith1) in the table. Next, we will follow the path to the node with the ID "Smith1" and traverse down every sub-element of the node. We will encounter an IDREF to "Smith2", and store ID ""Smith1" as source and "Smith2" as destination in the table. Some point later, we will encounter again an IDREF from "Smith2" to "central". Since the pair (central, Smith1) is already in the table, we terminate the traversal.

5. PERFORMANCE STUDIES

We implemented the mqX-scan operator in Java. The experiments are conducted on a Pentium III 866 MHz machine running Windows 98. All structures are kept in memory during the experiments. We use the XOO7 benchmark [4] to generate our XML documents, and Xerces Java Parser 1.4.3 [15] with SAX 1.0

[14] enabled to parse the documents. The XOO7 benchmark provides several parameters to change the size of an XML document both depth-wise and breadth-wise. We modify the XML document generator to associate IDs with attribute fields. IDREFs will point to randomly selected Ids. For each experiment, we run the document generator to produce a random XML document and stream the document to our operator. The query generator produces a set of queries on the document. The performance metric used is the time taken to parse the document from the root until all the queries have been bound. The cost of creating the document and queries are not included in the metric. Every experiment is run 8 times and the average value is taken. Table 1 shows the parameters and the range of values used.

Table 1. Parameters used in experiments.

	Range	Default	Description
F	50 – 1000K	231K	File size
P	100 – 100K	1000	Number of queries
R	50 – 2000	Nil	Number of IDREF references
D	2 – 8	6	Depth of queries
W	20% to 80%	0%	Percentage of queries containing a Kleene-star wildcard (_*).
M	0 or 1	0 (Relevant)	Parameter to generate relevant queries or mixed queries.
θ	0 or 1	0	Zipfian distribution

F denotes the file size used in the experiments. We vary this parameter to study the scalability of the system in terms of file size under the context of supporting many queries in a single pass of the XML document. The file size ranges from 50KB to 1000 KB. We found that when the file size reaches 24MB, the time required for parsing 500 queries is more than two hour.

Q is the number of queries in the system. Varying Q allows us to measure the scalability of our new operator in terms of number of queries issued. Q is fixed at 1000 in the comparative experiments.

D denotes the maximum level of a query. D is set to an average of 6 in most of the experiments. When a query involves _* wild card, the query level may differ from the level of the document. This is because the document generator always starts from the root while the query generator may begin a query with _*.

The rest of the parameters are used to shape the query workload. W is the percentage of queries in the set of the queries generated that will contain a Kleene_Star wild card (_*). This parameter is varied to evaluate the scalability and robustness of mqX-scan when dealing with wildcard. Another parameter, R, is varied to test the scalability and navigational functionality of mqX-scan when the documents contain ID and IDREF references.

M is the percentage of irrelevant queries. This parameter simulates the situation where users issue queries that do not contain any element name found in a document that is being parsed. When M is 0, the queries generated by the query generator are all relevant queries. That is, all the element names in a query can be found in that document. When M is 1, the queries generated will contain irrelevant queries based on a ratio in the setting when distributing the relevant queries and irrelevant queries in the queries generated. For example, if the ratio is 1:1, it means the queries generated will contain 50% relevant queries and 50% irrelevant queries.

We use θ as a parameter of Zipf distribution to determine the skewness of the choice of element names at each level in the query generation based on an input DTD. When θ is 0, each element name in the query is selected randomly from the sets of element names allowed at its level with a uniform distribution. When θ is 1, the choice is highly skewed. The level of skewness is set to 0.8.

5.1 Sensitivity Experiments

In this section, we investigate the performance of the mqX-scan operator under the following settings:

- No KP index is built (*mqX-1*).
- A KP index is built but no IDREF traversals (*mqX-2*).
- A KP index is built, together with IDREF traversals (*mqX-3*).

Since mqX-scan is applicable to tree-structured data as well, we examine the scalability of our operator for tree-structured data which do not require the construction of the KP index. This is because such an index provides no benefits for tree-based RPE evaluation. This leads to the first setting: *mqX-1*

When a document contains IDs and IDREFs, we need to evaluate the queries under the environment where IDREF traversals must be carried out. In such a situation, we have the third setting: *mqX-3*. The second setting: *mqX-2*, aims to show the cost incurred when the KP index is built but with no IDREF traversals.

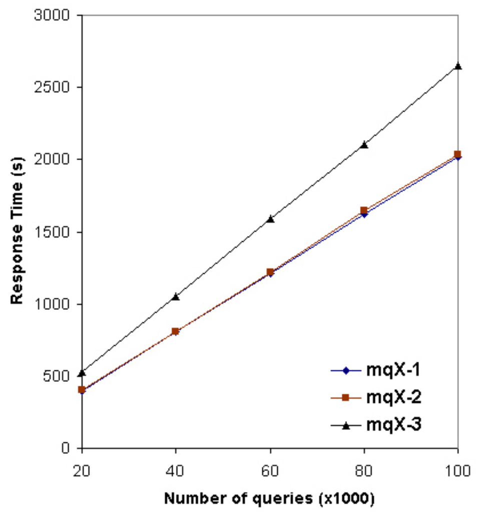

Figure 9. Effect of Number of Queries.

<u>Effect of Number of Queries</u>: F, D, W, M, θ: Default

In this experiment, we vary the number of RPE queries from 20000 to 100000. We use a uniform distribution to randomly select the element names in the query from a set of element names. The result of the study is shown in Figure 9. First, we observe that all the total response time for all the three queries increases linearly with the number of queries. Second, we see that *mqX-3* takes the longest time to bind all the values to the queries. This is caused by the extra traversals required by IDREFs. The cost of building the KP index is small, and it is nearly equivalent to the time taken by *mqX-1* to complete the bindings.

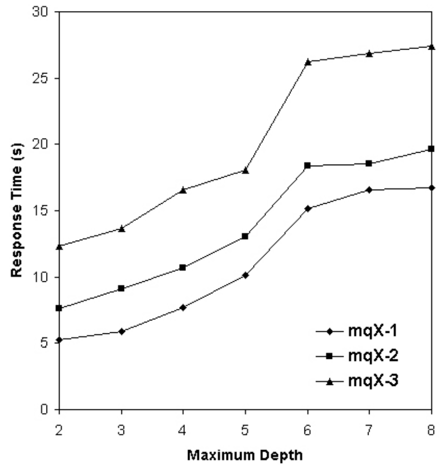

Figure 10. Effect of Query Depth.

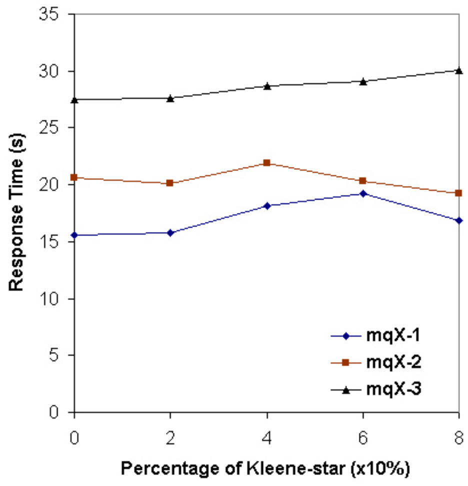

Figure 11. Effect of Kleene-star.

<u>**Effect of Depth:**</u> **F, P, W, M, θ: Default**
Next, we vary the depth of the queries from 2 to 8. Figure 10 shows that when the depth of the queries increases, the response time also increases. This is expected since there is an increase in the number of paths to traverse. Again, *mqX-3* has the largest response time due to the IDREFs traversals. However, we note the mqX-scan operator still gives fairly good performance under all the three settings even when the depth reaches 8.

<u>**Effect of Wild Card**</u>**: P, D, M, θ: Default, F=231KB**
This experiment evaluates the effect of Kleene-star wild card occurring in queries. W=2 indicates that two out of ten queries have a Kleene-star. Figure 11 shows that *mqX-3* has the largest response time. However, the performance of *mqX-1, mqX-2,* and *mqX-3* are not sensitive to the occurrence of Kleene-star wild card. Overall, the performance of mqX-scan operator does not degrade even when large number of queries containing Kleene-star wild card is issued.

5.2 Comparative Experiments

In this set of experiments, we compare the performance of mqX-scan (*mqX-2* and *mqX-3*) with X-scan on both skewed and uniform query distributions. To provide a fairer comparison with *mqX-2* and *mqX-3*, we implemented two versions of X-scan: *X-2* and *X-3*. The former disables IDREFs traversals while the latter involves IDREFs traversals.

<u>**Effect of File Size**</u>**: P, W, M: Default**
Figure 12 shows the results when the file size varies. Each scheme is appended with (U) for uniform distribution, and (S) for skewed distribution. It is clear that *mqX-2* outperforms *X-2* because the former avoids repeated parsing of the same document. We note that *mqX-2* is slower when the query distribution is skewed. This is expected as the waiting time for some queries will be increased.

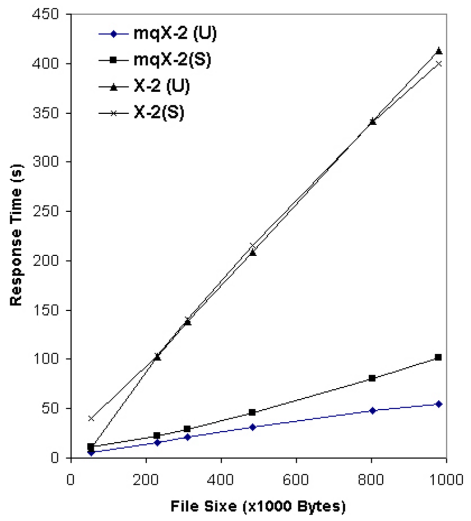

Figure 12. Effect of File Size.

<u>**Effect of Ratio of Relevant vs. Irrelevant Queries:**</u>
P, D, W, M: Default, F=56KB
We also implemented a version of mqX-scan with query indexing, *mqX-3(index)*. The latter uses techniques similar to XFilter [1] whereby an index is built on the queries. Figure 13 shows that *mqX-3* gives the best performance while the response time for *mqX-3(index)* grows exponentially due to repeated probing of the elements in the hash table. We observe that *mqX-3(index)* is superior when the ratio of irrelevant queries is very high. This is due to the absence of keys found in the hash table, eliminating the need for traversals. In contrast, graph traversals are required in *mqX-3* and *X-3* although the queries contain irrelevant names.

<u>**Effect of Number of Queries and IDREFs Traversals**</u>
F, P, D, M: Default, W=4
Next, we compare the cost of building the KP index and the *response* time when IDREFs traversals are involved. Each scheme is appended with (_*) when wild card is used. From Figure 14, we see that *mqX-2* outperforms *X-2* when the number of queries increases. *X-2* incurs high cost in building the structural index

[11] by repeatedly parsing the same document. The occurrence of the Kleene-star in either uniform or skew distributions does not affect the response time as shown in Figures 14 and 15. Again this explains that the performance of the mqX-scan operator is not affected by the occurrence of Kleene star wild card.

Figures 16 and 17 show that the presence of IDREFs causes the response times for *X-2* and *X-3* to increase dramatically. However, *mqX-2* and *mqX-3* exhibit stable response times. This demonstrates that when more IDREFs are encountered, more paths need to be traversed. Note that the difference in the response times between *mqX-2* and *mqX-3* reflects the time taken to traverse the IDREFs, which is negligible. It is also clear from the graphs that the distribution has no effect on the response time.

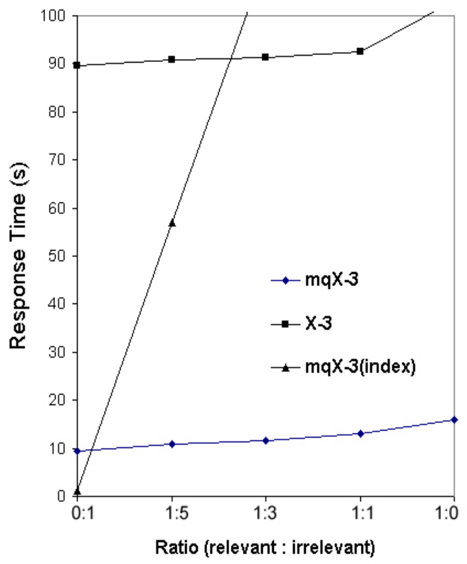

Figure 13. Relevant vs. Irrelevant Queries.

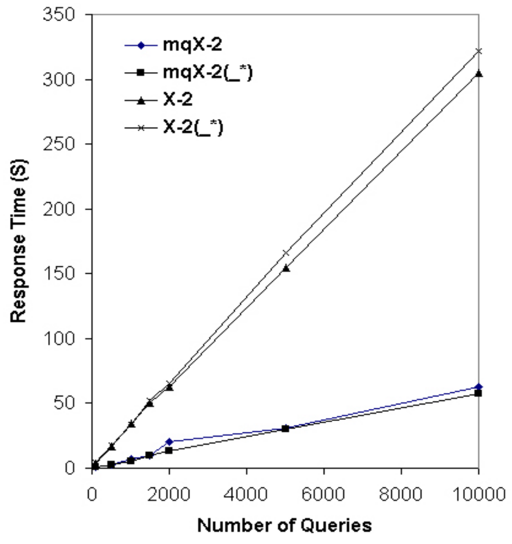

Figure 14. Effect of Varying Number of Queries. (Uniform Distribution).

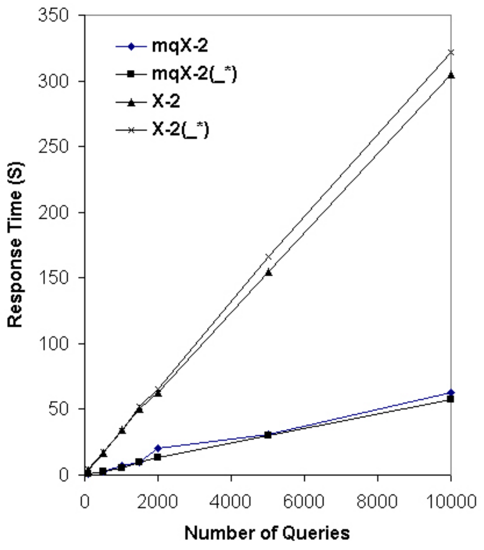

Figure 15. Effect of Varying Number of Queries. (Skewed Distribution).

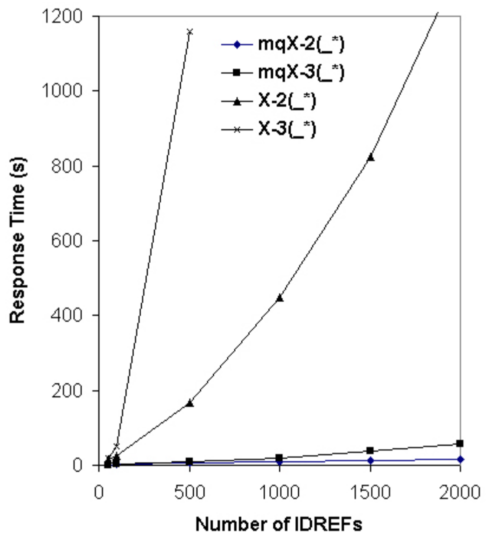

Figure 16. Effect of IDREFs (Uniform Distribution).

6. CONCLUSION

In this paper we have presented the mqX-scan operator that evaluates multiple queries using a single pass of XML documents. The proposed operator is pipelined and produces bindings as the XML document streams into the system. The results of our experiments demonstrate the scalability of mqX-scan in terms of number of queries and file size. We also note that mqX-scan can handle IDREFs and can be reused in SDI applications *without* the need to index the queries.

We found that by using the indexing technique proposed in Xfilter, the mqX operator performs badly due to the heavy probing and searching required when the percentage of relevant queries is high. One possible enhancement would be to tune the mqX-scan operator such that it becomes sensitive to the fluctuation of the composition of queries. A future direction would be to improve the efficiency of the indexing technique and group optimization for RPE queries when similar RPE queries are issued to a streaming XML document.

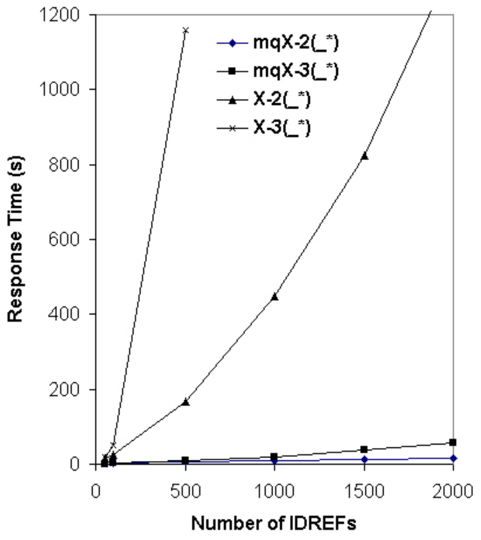

Figure 17. Effect of IDREFs (Skewed Distribution).

7. REFERENCES

[1] M.Altinel, M.J.Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information, VLDB, 2000.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom and J. Wiener. The Lorel Query Language for Semistructured Data, International Journal on Digital Libraries, Vol 1(1), pp. 68-88, 1997.

[3] D.Chamberlin, D.Florescu, J.Robie, J.Simon, M.Stefanescu. XQuery: An XML Query Language, W3C Working Draft, 2001.

[4] S.Bressan, G.Dobbie, Z.Lacroix, M.L.Lee, Y.Li, U. Nambiar and B.Wadhwa: XOO7: Applying OO7 Benchmark to XML Query Processing Tools, 10th ACM CIKM, 2001.

[5] S. Babu and J. Widom. Continuous Queries over Data Streams. Stanford University Tech. Report, 2001.

[6] J.Clark and S.DeRose. XML Path Language (XPath), W3C Technical Report, 1999.

[7] J. Chen, D. DeWitt, F. Tian and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases, ACM SIGMOD, 2000.

[8] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. WebDB Workshop, 2000.

[9] S. Cluet and J. Simeon. YATL: A functional and declarative language for XML. Working draft, 2000.

[10] A. Deutsch, M.F. Fernandez, D. Florescu, A. Levy, D. Suciu. A query language for XML. 8th WWW Conference, 1999.

[11] Z.G. Ives, A.Y. Levy and D.S. Weld. Efficient Evaluation of Regular Path Expressions on Streaming XML Data. Technical Report, University of Washington, 2000.

[12] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions, VLDB, 2001.

[13] J. Robie. The design of XQL, 1999. http://www.texcel.no/whitepapers/xql-design.html

[14] SAX 1.0: The Simple API for XML. http://www.saxproject.org/

[15] Apache XML Project. The Xerces Java XML parser. http://xml.apache.org/ xerces-j/index.html