Efficient Path Query Processing on Encoded XML

Yi Chen
University of Pennsylvania
Philadelphia PA 19104
yicn@cis.upenn.edu

George A. Mihaila
IBM T.J. Watson Research Center
Hawthorne, NY 10532
mihaila@us.ibm.com

Sriram Padmanabhan
IBM Silicon Valley Labs
San Jose, CA 95141
srp@us.ibm.com

Susan B. Davidson
University of Pennsylvania
Philadelphia PA 19104
susan@cis.upenn.edu

ABSTRACT

As XML has become a popular format for information exchange, the efficient processing of broadcast XML data on a constrained device (for example, a cell phone or a PDA) becomes a critical task. In this paper we present the EXPedite system: a new model of data processing, which "migrates" the power of the data-sending server to receivers for efficient processing. It consists of a general encoding scheme for servers, and streaming query processing algorithms on encoded XML stream for client devices. Preliminary experiments show the impressive performance of the EXPedite system.

1. INTRODUCTION

XML has become a popular format for information exchange. Consider, for example, a broadcasting system where the server broadcasts information as an XML stream, and each receiver extracts information of interest for further processing. Since receivers are often constrained devices (for example, cell phones or PDAs), processing XML data efficiently with limited computing ability and memory becomes a core technical challenge.

As a sample application, about 146 organizations recently formed a global forum called TV-Anytime [2], with the vision of using digital broadcasting as an opportunity to provide interactive TV services. For example, suppose that you are interested in the NFL game between the Eagles and the Giants. Using the current TV environment, you must look through numerous channel-by-channel program listings to find out that it will be shown on channel 5 at 9:30 tonight. You can then set your VCR to record it. Now imagine your VCR as a personal program guide which takes as input a personalized query, for example the NFL game between the Eagles and the Giants. The VCR then tracks down the program or programs corresponding to your view preference and intelligently records them. To enable this personal TV portal, TV-Anytime proposes to broadcast meta-data about program schedules, so that home devices (e.g. VCRs) can make content-based retrieval of the meta-data and find out the time and channel of programs that a user is interested in. One proposed solution is to use MPEG-7 [1], an XML schema specialized for audio-visual applications, as the format for meta-data.

Another application is a publish-subscribe system [16, 12, 10]. Since ender users' interests are very diverse, often it is too expensive for the server to evaluate individual queries for millions of subscribers. It is more realistic to send out half-processed information and then let users do a small amount of post-processing according their specific need. For example, consider a stock publish-subscribe system, where each user could be interested in tracking the stock of a distinct set of companies. The server could broadcast stock information to a group of users who have similar interests (e.g. hi-tech stocks), and let each user filter out specific information according to his/her need.

In both information exchange applications, receivers must be able to process broadcast XML data using an XML query language(for example, XPath [7]) in a streaming fashion, since the information arrives continuously, and may be either too large to fit in limited storage space or need real-time response. Lightweight XML query processing in a streaming fashion on constrained devices brings a big technical challenge.

Before we discuss how to address this challenge, first let us review current data processing models. Traditionally, there have been two models for data processing: database management systems, and stream data processing. A database system has powerful computing ability, large storage space, and various auxiliary data structures(for example, indexes and materialized views) to improve query processing performance. On the other hand, stream processing is often performed on constrained devices with limited computing ability and space without any auxiliary data structures. There is therefore a big gap between the performance of a database system and a stream processing system. A natural question to ask is if we can combine these two models and have the best of both worlds. In this paper we present the Encoded XML Processing system, EXPedite, for information exchange applications, which bridges this gap by distinguishing between the abilities of the data-sending server and receivers, and "migrating" the power of the server to the receivers by including the information gathered by the server in the encoded data to speed up receivers' processing. It thus represents a new model of data processing in an information exchange environment.

In this paper, we propose to leverage the power of the data-sending server to preprocess XML data and achieve a lightweight workload at the data receivers. Many recent papers [17, 27, 13, 8, 3, 4, 6, 15, 26] show that in XML databases, query processing over an interval-based labeling scheme has a substantial performance improvement. In this paper, we explore the labeling scheme in information-exchange applications. We have designed an encoding scheme for the server such that the labels for XML nodes can be

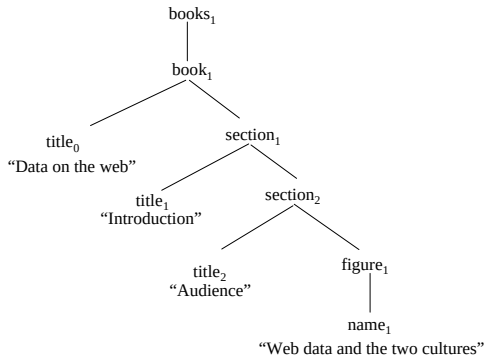


Figure 1: A sample XML data

computed easily from the encoded XML data by the data receivers, and therefore the query processing is very efficient. By encoding XML data on the server, EXPedite takes advantage of the processing power of the server, bridges the gap between database and data streaming processing, and enables substantial performance improvement on the constrained devices of data receivers. Furthermore, the query processing algorithm on XML labels in EXPedite has improved complexity over the algorithms in the literature.

The main contributions and organization of the paper are:

1. The EXPedite system utilizes an interval-based labeling scheme to enhance query processing performance for information exchange applications, which is traditionally used in XML databases. To the best of our knowledge, this is the first paper to propose an encoding scheme for XML data in order to speed up query processing.
2. A simple and effective XML encoding scheme ran by the server is proposed in section 2 to facilitate subsequent parsing and query processing on data receivers.
3. An algorithm on encoded XML streams that takes advantage of the labeling scheme for query processing on the data receivers is presented in section 3.2. The time complexity is $O(|Q||F|)$, where $|Q|$ and $|F|$ are the size of the query and encoded data, respectively, which improves the complexity $D^{|Q|}|Q||F|$, where D is the document depth, proposed in the literature [4].
4. Preliminary empirical study of EXPedite and several related systems in section 4 shows the substantial performance benefit of our approach.

2. BINARY ENCODING OF XML DATA

The EXPedite system is comprised of an encoder on the data-sending server, a parser and a query processor on data receivers. In this section, we will present the EXPedite encoder which takes a regular XML file and generates an encoded XML file, and the EXPedite parser which parses an encoded XML data.

2.1 Encoding XML data

The key idea of EXPedite is to allow data receivers to benefit from the information gathered by the data sender to improve performance. As observed in [17, 27, 13, 8, 3, 4, 6, 15, 26], the query processing performance in an XML database can be improved considerably by using an interval-based labeling scheme. The EXPedite encoder over the data-sending server applies an encoding scheme such that the labels of XML nodes can be computed easily to enable efficient query processing on receivers.

In this paper we do not distinguish between attributes and element nodes, and refer only to tags in the rest of the paper [1]. We distinguish two types of information in XML data: *structure information*, consisting of element tags and *value information*, consisting of text nodes, and encode them in different ways.

1. **Structural information** The code for a start tag T is a vector $< flag, t, size, depth >$.
   $flag$ is set to 0 to denote that this is structural information. A dictionary is built to map each distinct tag to a unique integer, and t is an integer corresponding to the tag T; $size$ is the byte size of the encoded subtree of T; and $depth$ denotes the number of tags on the path from the root to tag T, inclusively.
2. **Value information** The code for a value is a vector $< flag, length, text >$.
   $flag$ is set to 1 to denote that this is value information; and $length$ denotes the length of the text.

Since the encoding of a start tag records the size of an XML node (subtree), we can locate the position of the matching end tag, and therefore do not encode end tags.

Example 2.1: As an example, let us look at how to encode the sample XML data in figure 1, where we use subscripts to distinguish between nodes with the same tag. Suppose that we map tag $books$ to integer 1, $book$ to 2, $title$ to 3, etc. The first node tagged by $title$ ($title_0$) is encoded as a vector $< 0, 3, 15, 3 >$, and the value "Data on the Web" is encoded as $< 1, 15,$ "Data on the Web" $>$. ■

The algorithm for the EXPedite encoder is presented in algorithm 1. The encoded XML file will be sent to data receivers along with the dictionary of tag-to-integer mapping as the header of the encoded XML data.

Notice that we compute the length of element nodes and values and record them in the encoded data. This information will be used in subsequent processing to achieve substantial performance speedup, as will be discussed in section 2.2 and 3.2. Recording the length information entails a non-streaming encoding procedure. However, as we illustrated in the introduction, we distinguish between the role and the power of the data-sending server and those of the data receivers. The data-sending server has powerful computation abilities and often is the data generator (for example, in TV-anytime application), so the encoding operation does not need to be performed in a streaming fashion. On the other hand, as a data consumer, a data receiver has constraint computation ability and is infeasible to buffer huge data before processing, streaming processing information is necessary. Next we will discuss how to parse and query encoded XML data on the receivers in a streaming fashion.

2.2 Parsing encoded XML data

To parse encoded XML data, the EXPedite parser starts from the beginning of the encoded XML stream and retrieves the header containing the mapping between tags and integers. It then iterates

[1]The encoding scheme can be easily extended to differentiate attributes and element nodes.

Algorithm 1 Algorithm for EXPedite Encoder

```
1: function Encode
2: input: an XML file X
3: output: an encoded XML file F
4: A stack S to record the start position of each tag t in F
5: A hashtable M to map a tag T to an integer t
6: depth = 1
7: currPos = 0 {record the current position of F}
8: while !eof(X) do
9:    if SAX event = a start tag < T > then
10:      t = M(T)
11:      write(F, < 0, t, 0, depth >) {actual size will be calculated at
         end tag}
12:      push(S, currPos)
13:      depth++
14:      currPos = currPos + sizeof(code for tag)
15:   end if
16:   if SAX event = an end tag < /T > then
17:      startPos = pop(S) {locate the position of corresponding < T >
         in F}
18:      size = currPos-startPos+1
19:      seek(startPos)
20:      write(F,size) {fill size in the code for < T >}
21:      seek(end of F)
22:      depth--
23:   end if
24:   if SAX event = character data C then
25:      write(F, < 1, lengthof(C), C >)
26:      currPos = currPos + sizeof(code for value)
27:   end if
28: end while
29: return F
```

over the rest of the file. If the first integer in the remainder of the encoded data is 0, the parser recognizes that it is a tag and retrieve the vector $< flag, t, size, depth >$. Otherwise it is a value and the vector $< flag, length, text >$ is retrieved. The parser proceeds in this way until the end of the encoded stream.

When parsing encoded XML stream, we can compute the interval based labels of the XML nodes,$< start, end, depth >$, easily since the current position is the $start$ and $end = start + size$. Following [17, 27, 13, 8, 3, 4, 6, 15, 26] structural relationships between nodes can be efficiently determined from the labels:

1. n_1 is an ancestor of n_2 if and only if $n_1.start < n_2.start$ and $n_1.end > n2.end$.
2. n_1 is the parent of n_2 if and only if n_1 is an ancestor of n_2 and $n_1.depth + 1 = n_2.depth$.

There are several advantages to parsing encoded XML data over parsing regular XML data.

1. Integer processing is very efficient.
2. The labels for XML nodes can be obtained easily, which enables efficient query processing.
3. The $size$ information can function as a stream index for XML data (SIX) as discussed in [12], and therefore be used to skip subtrees whose content are irrelevant to our interest.
4. By including the $length$ of each text value before the value, we retrieve value information lazily.

In the next section, we will discuss in detail how the EXPedite query processor benefits from the encoded information and achieves good performance.

3. QUERY PROCESSING OVER ENCODED XML STREAMS

This section presents the EXPedite query processor. We begin with some preliminaries on XPath queries, and then present the query processing algorithms in detail.

3.1 Preliminaries: XPath queries

In this paper we focus on a subset of XPath queries, which contain child axis "/", descendant axis "//" and name tests. This type of query is commonly used as a building block for more complex XPath and other XML query languages. A query can be represented as a linear path, therefore is named as *path query*. For example, query */books/book//section//title* is shown in figure 2(b). In this representation, a node is created for each tag and the return tag is indicated by a box. For the incoming edge of a node, a single line represents a child axis, and double lines represent a descendant axis.

To process a path query efficiently on an encoded XML stream, we first encode the query as follows: From the header of the encoded XML stream, we get the dictionary that maps an XML tag to an integer. Then we replace every tag in the XPath query with its corresponding integer, and keep the value predicate as it is. For example, a query $/books/book/title[text() =$ "Data on the Web"] is encoded as $/1/2/3[text() =$ "Data on the Web"].

3.2 A query processing algorithm

The EXPedite query processor takes an encoded XML stream F, where each structural node n is encoded as $< t, size, depth >$ (we ignore $flag$ since only structural information is considered), and an encoded XPath query Q as input, and outputs the encoded fragment in F that matches Q.

The query processing algorithm $PathQuery$ is presented in algorithm 2. The basic idea of the algorithm is the following: we build a stack for each node in the query tree. The stacks also form a tree corresponding to the query tree. The stacks store the information of XML nodes which are solutions to the subquery from the root to the corresponding query node of the current stack. Therefore the nodes in the stack corresponding to the return node in the query are the query solutions.

A node is popped out of stack if it is out-of-region. We notice that each (encoded) XML node has a region between its start tag and end tag. If the byte we are currently processing is within the region of a node, the node is called an "active node", otherwise it is a "stale node". Since a stale node can no longer contribute to the query pattern match, the stacks store only the active nodes.

The query processing over XML stream using the stack proceeds as follows: When we process a node n in an XML stream F, if it matches stack S_t (i.e. its tag matches a node t in Q, line 10) and is a "qualified" node (line 13), we push it onto S_t. A node n matching S_t is *qualified* if either its depth matches the axis of the root stack, or there is some XML node m in the parent stack $S_{parent(t)}$ such that m and n satisfy the axis (child or descendant) relationship specified for t and $parent(t)$ in Q. As we advance the XML stream to meet n, we pop out stale nodes from stacks according to the position of n (lines 11-12). If n's tag matches the return stack (line 14) and is qualified, it is output as part of the query result. Let us first look at an example.

Algorithm 2 Algorithm for EXPedite Query Processor

```
 1: function PathQuery
 2: input: an encoded XML stream F
 3: input: an encoded XPath query Q
 4: output: fragments of F matching Q
 5: Build a stack S_t for each encoded tag t in Q
 6: currPos = 0 {Record the current position in F}
 7: while !eof(F) do
 8:    read next node n :< n.t, n.size, n.depth > in F, currPos++
 9:    t = n.t
10:    if t matches a tag in Q then
11:       popStack(S_t, currPos)
12:       popStack(S_parent(t)), currPos)
13:       if ( isRoot(t) and (descendant-axis(t, parent(t)) or n.depth==1) )
          or (( !empty(S_parent(t)) and (descendant-axis(t, parent(t)) or (
          child-axis(t, parent(t)) and n.depth == top(S_parent(t)).depth
          + 1))) {Check if n is qualified to push.} then
14:          if isReturn(t) then
15:             output subtree of n
16:          else
17:             push(S_t, < currPos, currPos + n.size, n.depth >)
18:          end if
19:       end if
20:    end if
21: end while
22:
23: procedure popStack{Remove stale nodes in a stack}
24: input: a stack S
25: input: an integer p
26: while !empty(S) and top(S).end < p do
27:    pop(S)
28: end while
```

Example 3.1: Consider a path query Q_1 = */books/book//section//title* on the XML data in figure 1. To be concise, we re-draw the XML data in figure 2(a), excluding value information. The query Q_1 is represented as a twig in figure 2(b). For clarity, rather than using encoded nodes we use full tag names. We also use subscripted tag names to represent nodes in the XML stream and on the stack.

First, we build a stack for each tag in Q_1. Then we read the encoded XML stream sequentially. Since $books_1$ has depth 1 and matches the root stack S_{books}, it is pushed to the stack. For node $book_1$ which matches stack S_{book}, since the top element $books_1$ in stack S_{books} is the parent of $book_1$, which satisfies the child-axis "/" in Q_1 (line 13), $book_1$ is qualified to be pushed onto stack S_{book}.

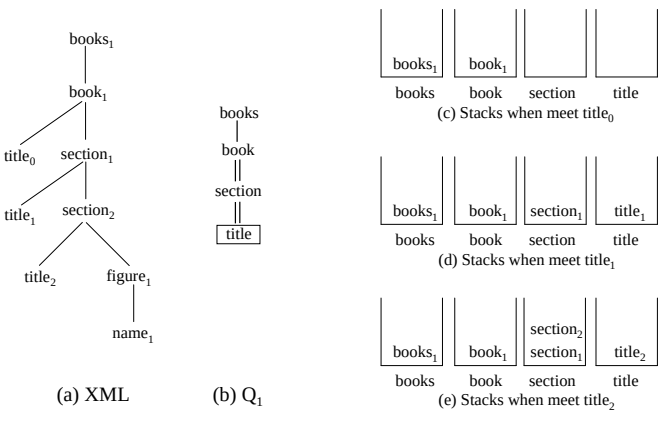


Figure 2: Example of path query processing

stack $S_{section}$ is empty, it is not qualified to be pushed. The current state of the stacks is now shown in figure 2(c). Then we push $section_1$ to stack $S_{section}$. When we process $title_1$, we notice that its parent stack $S_{section}$ is not empty and $title$ is connected with $section$ by descendant-axis "//" in Q_1, so it is a qualified node. Since stack S_{title} corresponds to the query return node, the subtree of $title_1$ is output. The snapshot of the stacks at this point are shown in figure 2(d). Similarly, $title_2$ is also a qualified node and is output, as shown in the snapshot of the stacks in figure 2(e). ■

3.3 Analysis

Theorem 3.2: Algorithm $PathQuery$ is correct. ■

The proof is based on induction over the query path starting from the root node. It is omitted here for reasons of space.

Theorem 3.3: The complexity of algorithm $PathQuery$ is $O(|Q||F|)$, where $|F|$ and $|Q|$ is the size of the XML stream and query, respectively. ■

Theorem 3.4: The storage requirement for $PathQuery$ is bounded by $O(|Q|D)$, where $|Q|$ is the size of the query, and D is maximum number of repeated tags in a root-to-leaf path in the tree representation of the XML stream. ■

Algorithm 2 is inspired by the $PathStack$ algorithm of [4]. However there are several important optimizations.

First, we only push qualified nodes and have fewer computations (line 13). Each pushed node must be of a solution to the subquery from the query root to the query node corresponds the current stack. On the other hand, the $PathStack$ algorithm in [4] pushes every node as long as its name matches a stack(for example, the $title_0$ in the above example), and therefore requires unnecessary pushes and pops.

Second, since only qualified nodes are pushed in $PathQuery$ algorithm, a node qualifies to be pushed onto the return stack is a query solution. In algorithm $PathStack$, when a node matches the return stack, it needs to check nodes in all the stacks to determine whether or not there exist a pattern match to "qualify" current node for a valid query answer, which requires time complexity $O(|F||Q| * D^{|Q|})$.

Third, our stack structures are simpler. Since we are able to output query solution without enumerating pattern matches, we do not need to maintain a pointer from a node in one stack to a node in its parent stack, as in [4].

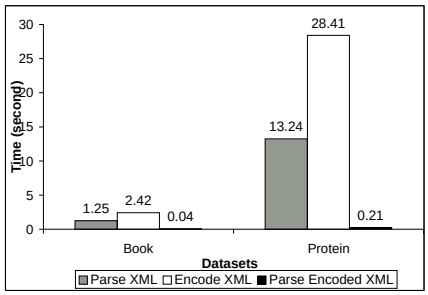
4. EXPERIMENTAL EVALUATION

We have implemented EXPedite in C++. The XML parser used is the Xerces SAX2 parser. In this section, we present some preliminary performance study of this implementation.

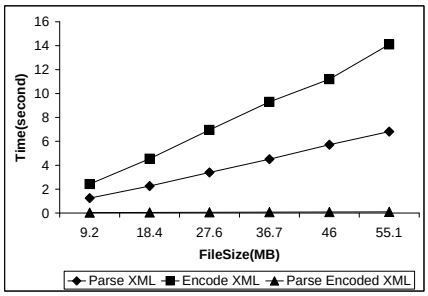
4.1 Experiment setup

All experiments were conducted on a Pentium III 1.5GHz machine with 512MB memory, running the Redhat 9 distribution of GNU/ Linux(kernel 2.4.20-8).

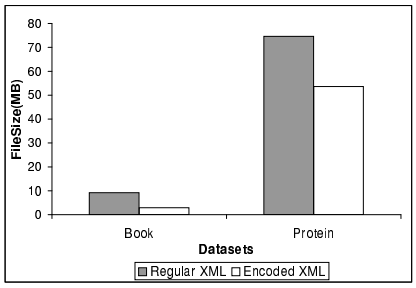
We compare EXPedite with several systems which support XPath



(a) Parsing/Encoding Time for Different Datasets



(b) Parsing/Encoding Time as File Size Increases



(c) XML and Encoded XML File Size

Figure 3: Performance for XML Encoding

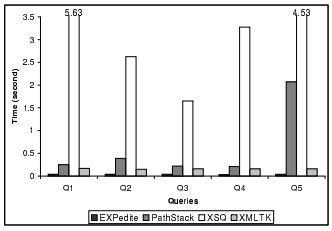
Book Dataset

Q_1	/book//section//title
Q_2	/book//section/figure/image/@source
Q_3	//title
Q_4	/book//section/p
Q_5	//section//figure/title

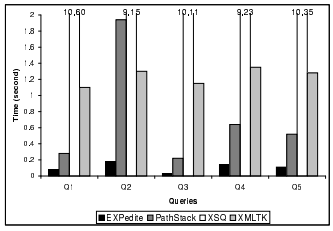
Protein Dataset

Q_1	/ProteinDatabase//protein/name
Q_2	//ProteinEntry//reference/refinfo//xrefs//xref//db
Q_3	//organism/source
Q_4	//refinfo/citation/@type
Q_5	/ProteinDatabase//ProteinEntry/classification/superfamily

Figure 4: Query sets



(a) *Book* dataset (9 MB)



(b) *Protein* dataset (75 MB)

Figure 5: Query Execution Time

query processing: *Path/TwigStack* [4]: the start-of-art XML query processing using labeling schemes, *XMLTK* [12]: a streaming path query processor using DFAs and *XSQ* [22]: a streaming XPath query processor using transducers. Of the above, XMLTK supports the same XPath subset as EXPedite, while Path/TwigStack and XSQ additionally support XPath fragments containing predicates and are more powerful. The release versions of other XML query streaming processors such as XSM [19], YFilter [9] and XTrie [5] are not available at the time being.

We conducted experiments on two datasets. The first dataset is a real dataset from the International Protein Sequence Database [11]. The DTD of the protein data is a tree containing 66 tags. The text values in this dataset are large (e.g. a protein sequence). The second dataset is a synthetic dataset generated by IBM's XML Generator [14]. The input DTD is the *Book* DTD from the XQuery use cases [25], which is cyclic and contains 12 tags. The generated data values are small. We apply the default settings of XML Generator for all the parameters except that *NumberLevels* is set to 20 and *MaxRepeats* is set to 9.

4.2 Performance of XML encoding

Figure 3(a) shows the performance of a SAX2 parser, the EXPedite encoder and the EXPedite parser on two datasets. As we can see, the encoding time is around twice the SAX parsing time. Note that the encoding time is comprised of parsing and encoding, so the parsing time is a lower bound for the encoding time. We also see that parsing encoded data costs far less than SAX parsing of XML data. We can see that the compute power required for parsing encoded XML data is very small, hence suitable for constrained devices.

Figure 3(b) shows that the time for parsing XML, encoding XML and parsing encoded XML data increases linearly with the XML file size. We can see that the cost of parsing encoded data increases by very little.

Figure 3(c) shows that the encoded XML data is usually smaller than the original XML file, a side effect of EXPedite which is very beneficial in the data exchange scenario.

4.3 Query processing time

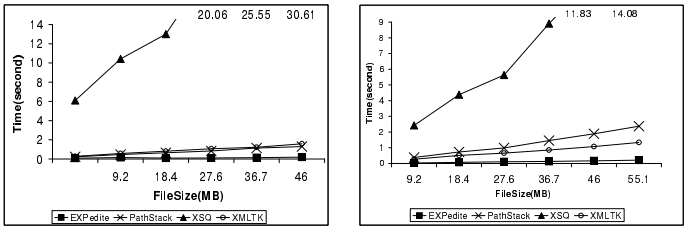
Next we compare the performance of XPath query processing on regular XML data with the EXPedite query processor on encoded XML data. The queries we tested are listed in figure 4.

In this section, we exclude the parsing time from the execution time in order to compare the query processing performance. As shown in section 4.2, the performance difference between EXPedite and other XML streaming processing systems would be more dramatic if we reported the overall time for parsing and query processing.

Figure 5 reports the query execution time of different XML query processing systems for the *Book* and *Protein* datasets over different queries. As we can see, processing encoded XPath queries on encoded XML data is much faster than regular XML query processing. The performance of EXPedite is also very stable: It performs well on recursive and non-recursive data, as well as simple and complex queries.

4.4 Scalability of query processing time

We duplicated the *Book* dataset between 2 and 6 times to get experimental datasets to test the scalability of query processing on different systems. Figure 6 reports the query processing time for increasing sizes of XML data on Q_1 and Q_2 respectively. As we can see, as the file size increases, the execution time of EXPedite increases much more slowly than other approaches.



(a) Q_1 (b) Q_2

Figure 6: Query Execution Time as File Size Increases

5. RELATED WORK

Related work includes XML query processing in a database system, streaming XML query processing and XML compressor.

There has been a great deal of recent work on XML query processing. One class of XML query processing is based on the $< start, end, level >$ label scheme of XML data [17, 27, 13, 8, 3, 4, 6, 15, 26], when the XML data is stored in a relational database. In this paper we designed an XML encoding scheme such that traditional query processing techniques based on this labeling scheme for XML databases can be used on XML data in an information exchange applications.

Another class of XML query processing is performed in a streaming fashion. *XMLTK* [12] is a streaming path query processor using a DFA (Deterministic Finite Automaton) constructed lazily. *XSQ* [22] process XPath queries on streaming XML data using a hierarchical pushdown transducer (HPDT). Another transducer-based approach, the XML streaming machine *XSM*, is presented in [19] to answer XQueries without descendant axes. [21] presents *SPEX*, a streamed evaluation of regular path expressions with qualifiers and backward navigation (XPath-like) against XML streams. Yfilter [9] and XTrie [5] discuss how to filter a large number of XML queries efficiently in a publish-subscribe system.

There are many works on XML Binary format proposed in W3C Workshop on Binary Interchange of XML Information Item Sets [23], and XML data compression, including XMill [18], XGrind [24] and XPress [20], where the goal is to save parsing time and transmission bandwidth. EXPedite focuses on speed up query processing, and can have compression applied to the encoded data to further reduce size if necessary.

6. CONCLUSIONS

This paper presents the EXPedite system: a novel model of data processing in an information exchange environment. We proposed a simple and general encoding scheme of XML data for data-sending servers, and presented algorithms for data receivers to efficiently process path queries over broadcasted encoded XML streams. The proposed query processing algorithm based on an XML labeling scheme improves the known complexity in the literature. Experiments show the substantial performance benefits of processing encoded data using EXPedite compared to regular XML stream processing.

Currently the EXPedite encoder supports XML elements and character data, and the query processor supports XPath queries containing child axis and descendant axis. In the future we will extend the encoder to fully support XML 1.0 and extend the query processor to handle all the axes in the XPath language.

7. REFERENCES

[1] Moving Picture Experts Group (MPEG), 2003. http://www.chiariglione.org/mpeg/index.htm.

[2] TV-Anytime Forum, 2003. http://www.tv-anytime.org.

[3] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of ICDE*, 2002.

[4] N. Bruno, N. Koudas, , and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of SIGMOD*, 2002.

[5] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proceedings of ICDE*, 2002.

[6] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents . In *Proceedings of VLDB*, 2002.

[7] J. Clark and S. DeRose. XML Path language (XPath), November 1999. http://www.w3.org/TR/xpath.

[8] D. DeHaan, D. Toman, M. Consens, and M. T. Ozsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proceedings of SIGMOD*, 2001.

[9] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *TODS*, 28(4):467–516, 2003.

[10] Y. Diao and M. J. Franklin. Query Processing for High-Volume XML Message Brokering. In *Proceedings of VLDB*, 2003.

[11] Georgetown Protein Information Resource. Protein Sequence Database, 2001. http://www.cs.washington.edu/research/xmldatasets/.

[12] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata . In *Proceedings of ICDT*, 2003.

[13] T. Grust. Accelerating Xpath location steps. In *Proceedings of SIGMOD*, 2002.

[14] IBM. XML Generator, 1999. http://www.alphaworks.ibm.com/tech/xmlgenerator.

[15] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *Proceedings of ICDE*, 2003.

[16] L. V. S. Lakshmanan and S. Parthasarathy. On Efficient Matching of Streaming XML Documents and Queries. In *Extending Database Technology*, pages 142–160, 2002.

[17] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *The VLDB Journal*, pages 361–370, 2001.

[18] H. Liefke and D. Suciu. XMill: an efficient compressor for XML data. In *Proceedings of SIGMOD*, 2000.

[19] B. Ludascher, P. Mukhopadhayn, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proceedings of VLDB*, 2002.

[20] J.-K. Min, M.-J. Park, and C.-W. Chung. XPRESS: A queriable compression for XML data. In *Proceedings of SIGMOD*, 2003.

[21] D. Olteanu, T. Kiesling, and F. Bry. An Evaluation of Regular Path Expressions with Qualifiers against XML Streams. In *Proceedings of ICDE*, 2003.

[22] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *Proceedings of SIGMOD*, 2003.

[23] L. Quin. W3C Workshop on Binary Interchange of XML Information Item Sets . http://www.w3.org/2003/08/binary-interchange-workshop/Report.html.

[24] P. Tolani and J. Haritsa. XGRIND: A query-friendly XML compressor. In *Proceedings of ICDE*, 2002.

[25] W3C. XML Query Use Cases, 2003. http://www.w3.org/TR/xquery-use-cases.

[26] W. Wang, H. Jiang, H. Lu, and J. X. Yu. PBiTree Coding and Efficient Processing of Containment Joins. In *Proceedings of ICDE*, 2003.

[27] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of SIGMOD*, 2001.