# EXPedite: A System for Encoded XML Processing

Yi Chen
University of Pennsylvania
yicn@cis.upenn.edu

George A. Mihaila
IBM T.J. Watson Research Center
mihaila@us.ibm.com

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

Sriram Padmanabhan
IBM Silicon Valley Labs
srp@us.ibm.com

## ABSTRACT

As XML becomes an increasingly 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 in an information exchange environment, which "migrates" the power of the data-sending server to receivers for efficient processing. It consists of a simple and general encoding scheme for servers, and streaming query processing algorithms on encoded XML stream for data receivers with constrained computing abilities. Experiments show the impressive performance of EXPedite.

## Categories and Subject Descriptors

H.3.5 [**Online Information Services**]: Web-based services

## General Terms

Design, Performance

## Keywords

XML, binary encoding, XPath, query processing

## 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*CIKM'04,* November 8–13, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-874-1/04/0011 ...$5.00.

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 perform content-based retrieval on 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 [18, 14, 11]. Since end 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 to their specific need. For example, consider a stock publish-subscribe system, where each user is 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 needs.

In both applications, receivers must be able to process broadcast XML data using an XML query language (for example, XPath [9]) in a streaming fashion, since the data arrives continuously, and may either be 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 abilities, large storage space,

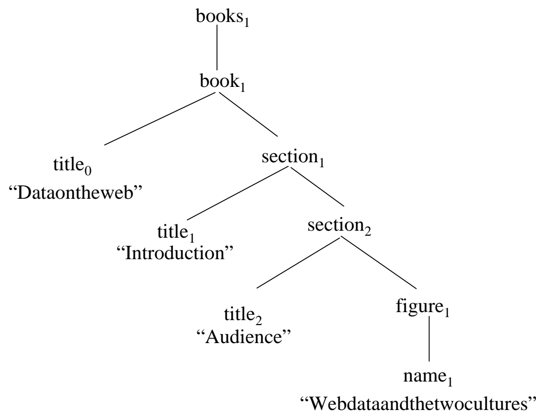



**Figure 1: A sample XML data**

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 that of 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 "migrates" 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 pre-process XML data and achieve a lightweight workload at the data receivers. Many recent papers [19, 30, 15, 10, 3, 5, 8, 17, 29] 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 computed easily from the encoded XML data by the data receivers, and therefore 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 data receivers which are constrained devices. Furthermore, the query processing algorithm on XML labels in EXPedite has improved complexity over the algorithms in the literature.

Our experimental evaluation shows that this approach is highly effective compared to regular XML query processing. For example, for a 75MB XML file of protein sequences, processing a complex XPath query on its encoded format using EXPedite takes about 1 second, while using a standard SAX parser on this XML data takes 8.2 seconds, and processing the query using a state-of-art streaming XML query processor requires an additional 19.9 seconds.

Besides the performance improvement, the encoding scheme enables us to process a broader class of XML queries than many existing work on XML stream query processing [24, 21, 23, 14, 11, 18]. We compare in detail the types of queries supported by these systems as well as their performance with our approach in section 4.

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 run 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 XPath query processing on the data receivers is presented in section 3.3.

4. We present a detailed empirical study of EXPedite and several related systems in section 4, which shows the substantial performance benefit of our approach.

## 2. BINARY ENCODING OF XML DATA

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

### 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 [19, 30, 15, 10, 3, 5, 8, 17, 29], the query processing performance in an XML database can be improved considerably by using an interval-based labeling scheme. The EXPedite encoder applies an encoding scheme such that the labels of XML nodes can be computed easily to enable efficient query processing at the 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.

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

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 data will be sent to data receivers along with the dictionary of tag-to-integer mapping as the header.

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 sections 2.2 and 3. 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, a data receiver has constrained computational ability and cannot buffer large amounts of data; streaming processing is therefore 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 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 easily compute the interval based labels of the XML nodes, $< start, end, depth >$, since the current position is the $start$ and $end = start + size$. Following [19, 30, 15, 10, 3, 5, 8, 17, 29] 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.

**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** $F$

3. The $size$ information can act as a stream index for XML data (SIX) as discussed in [14], and therefore be used to skip subtrees whose content is irrelevant to the query.
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 algorithm.

## 3.1 Preliminaries: XPath Queries

In this paper we focus on a commonly used subset of XPath queries, XPath expressions consisting of a sequence of location steps with child "/" and descendant "//" axes, branches (or predicates) "[]", and name tests. Since a query of this type can be represented as a small tree (twig) pattern, we call it a `twig query`. For example, the twig query $//section[//figure/name]/title$ is shown in figure 2(b). This query searches for XML nodes with tag *title* which have a parent *section*, and node *section* has a descendant *figure* which in turn has a child *name*. In this representation, we create a node for each tag and annotate the node with the

tag. The return node 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. If a node has more than one child then it is a *branching point*. If the return node is not a leaf, it is also a branching point. Twig queries contain branching points, whose subtrees can recursively contain branching points. Furthermore, if the query pattern is linear and the return node is the leaf, we call it a `path query`. For example, $/books/book//section//title$ is a path query.

To process an XPath query efficiently on an encoded XML stream, we first encode the XPath 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, an XPath query $/books/book/title[text() =$ "Data on the Web"] is encoded as $1/2/3[text() =$ "Data on the Web"].

Given an encoded XPath query $Q$ and an encoded XML stream $F$, a *pattern match* of $Q$ in $F$ is a set of nodes in $F$ such that there is a mapping from these nodes to nodes in $Q$, and the encoded XML nodes satisfy the location steps of the corresponding nodes in $Q$. The EXPedite query processor returns every encoded XML node in a query pattern match which maps to the return node.

Typically, to process a query in a database system we start with the most selective predicates (for example, the value predicates) in order to reduce the size of intermediate results. In contrast, to process a query over streaming data we proceed in a top-down fashion: find the nodes whose paths satisfy the structural pattern of the query, then test if the text values of those nodes satisfy the value predicates. Since checking value predicates for a given node is straightforward, we will focus on queries without value predicates (that is, structural matching) in the remainder of the paper.

## 3.2 Query Processor Features

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 basis of the EXPedite query processing algorithm is to evaluate queries based on XML labels $< start,end,depth >$. Many techniques [19, 30, 15, 10, 3, 5, 8, 17, 29] have recently been proposed for efficient query evaluation based on XML node labels for XML data stored in a database. Existing techniques follow a query decomposition approach. They first decompose a twig query into subqueries. Since a subquery (which could be a path query or a query only containing one axis) is much easier to process than a twig query, it can be evaluated very efficiently. After evaluating each subquery, they stitch together the intermediate results of subqueries using merge-joins to compute final solutions to the original twig query. This query decomposition approach is very efficient for processing XML data in databases since indexes and statistics can speed up subquery processing and join performance. However, it is not amenable to processing streaming data because:

1. Intermediate results of subqueries need to be buffered before the merge join, which may require a large buffer and delay response time.
2. To merge-join intermediate results efficiently, intermediate results for subqueries must be sorted, which may require blocking.

EXPedite query processor takes a novel approach to processing twig queries on encoded XML streams. Instead of decomposing a twig query to subqueries, it matches the twig pattern as a whole without generating intermediate results, and therefore is suitable for stream processing.

## 3.3 Query Processing Algorithm

The basic idea of the EXPedite query processing algorithm is as follows. Since XML data can be recursive and descendent axis can be present in a query, we build a stack for each node in the query to store multiple matches of a query node in an XML root-to-leaf path. When we meet an encoded XML node, we push it to a stack if it is a solution to the subquery from the query root to the corresponding query node of the current stack. At the same time, we pop out stale nodes from all stacks. 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 a query pattern match, it is popped out from the stack. Before we actually pop a stale node out of a stack, we check whether its corresponding XML node has a subtree match to the subtree of the corresponding query node. Therefore when we pop out a node from the stack of the query root, we can find out whether or not there is a match to the whole query.

In next sections, we will present the data structure and query processing algorithm.

### *3.3.1* Data structures

The EXPedite query processor takes an encoded XML stream $F$ and an encoded XPath query $Q$ as input, and outputs encoded fragments in $F$ that matches $Q$. This entails three different data structures:

1. An encoded XML stream $F$, where each structural node $n$ is encoded as $< t, size, depth >$.
2. An encoded XPath query $Q$ presented as a twig, where a node $x$ has a tag encoded as a code $t$.
3. A stack $S_x$ associated with each node $x$ in $Q$ as a buffer for processing. A node $m$ in stack $S_x$ records the information for XML nodes which are solutions to the subquery from the query root to $x$.

We have discussed the structure of encoded XML data; next we will discuss the query structure and the stack structure.

Throughout this section, we use the "." notation to refer to a component of a vector. For example, we use $n.t$ to retrieve the tag of an encoded XML node $n :< t, size, depth >$.

We represent an encoded XPath query $Q$ as a twig, and the following operations on $Q$ are supported:

- `tag`($x$): returns the tag associated with node $x$.
- `isReturn` ($x$): returns $true$ if node $x$ is the return node in $Q$, otherwise it returns $false$.
- `isRoot`($x$): returns $true$ if $x$ is the root of $Q$.
- `parent`($x$): returns $x$'s parent (which is connected to $x$ with either a child or a descendant axis) in $Q$.
- `asChild`($x_1$, $x_2$): returns $i$ if $x_1$ is the $i^{th}$ child of $x_2$.

- `desc-axis(`$x_1$`)`: returns *true* if $x_1$ is connected with its parent by a descendant axis "//".
- `child-axis(`$x_1$`)`: returns *true* if $x_1$ is connected with its parent by a child axis "/".

We support `push(`$S$`)`, `pop(`$S$`)`, `top(`$S$`)` and `empty(`$S$`)` operations on a stack. If we think of attaching $S_x$ to node $x$ in the twig $Q$, the stacks also form a twig. We can then refer to stack $S_{x'}$ as the `parent stack` of $S_x$ whenever $x' = parent(x)$ in $Q$. We also refer to the stack attached to the root of $Q$ as the `root stack` and the one attached to the return node of $Q$ as the `return stack`.

As we have discussed earlier, the elements in stacks are used to store information of the XML nodes which potentially contribute to a query pattern match. What information needs to be recorded for query processing?

First the label of an XML node, $< start, end,\ depth >$, is stored for comparing the relationship between XML nodes.

Second, since a query can contain branches, we may look for a match for a node in $Q$ with several children, say $t_1$ and $t_2$ for the purposes of illustration. When we meet an encoded XML node matching $t_2$, the node matching $t_1$ may be stale and have already been popped out of stacks. To avoid losing information about what predicates have been found, we need to record for each stack element whether or not each of its children has found a match, using a boolean array. We define the following operations on the array:

- `set(`$m$`, `$i$`)`: records that the $i^{th}$ child of $m$ has been found.
- `isComplete(`$m$`)`: returns *true* if each child of $m$ has found a match, otherwise returns $false$.

Furthermore, when an XML node matches the return node of the query, we may not be able to verify that it is a solution to the query since the encoded XML stream needs to be processed in order and the nodes matching branch condition may not have been seen yet. Therefore we associate with each stack element a set of possible solutions called `candidates`.

The meaning of candidates $c$ with respect to an element $m$ in a stack $S_x$ is that if $m$ is verified to be a match to the corresponding query node, then $c$ are solutions to the subtree of the query node. For example, consider the XML data in figure 2(a) and query $Q : //section[//figure/name]/title$ in figure 2(b). XML node $title_2$ is a possible solution, but can not be verified at the time it is being processed. So we associate it with node $section_2$ as a candidate. When we find out that $section_2$ has a child $figure_1$ and a grandchild $name_1$, we can determine that $title_2$ is a solution to $Q$. The action on $c$ depends on the status of $m$: If $m$ has found a match for each of its children in $Q$, then $c$ must be kept buffered (if $t$ is not the root of $Q$) or output (if $t$ is the root of $Q$); otherwise, $c$ is no longer valid with respect to $m$ and should be cleared. To process candidates, we define the following operations on an element in a stack:

- `buffer(`$m$`)`: starts buffering the current XML subtree and stores it as a candidate associated with $m$.
- `copy(`$m$`, `$m'$`)`: copies the candidates of $m'$ to $m$.
- `output(`$m$`)`: outputs the candidates associated with $m$ which have not been output yet.
- `clear(`$m$`)`: removes all candiates associated with $m$.

**Algorithm 2** Algorithm for EXPedite Query Processor

```
1: function TwigQuery
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_x for each encoded node x in Q
6: currPos = 0
7: while !eof(F) do
8:   read next node n in F, currPos++
9:   t = n.t
10:  for all x' in Q, in the bottom-up traversal order of Q do
11:    popStack(S_x', currPos)
12:  end for
13:  for all stacks S_x such that tag(x) = t in the bottom-up traversal order do
14:    if (isRoot(x) and (desc-axis(x) or (child-axis(x) and (n.depth = 1)) ) ) or ( !empty(S_parent(x)) and (desc-axis(x) or ( child-axis(x) and (n.depth - top(S_parent(x)).depth = 1)))) then
15:      m =< currPos, currPos + n.size, n.depth, Null, Null > {The array of child match and candidates are initiatated to be Null.}
16:      if isReturn(x) then
17:        buffer(m) {Buffer the subtree of n in F as a candidate of m}
18:      end if
19:      push(S_x,m )
20:    end if
21:  end for
22: end while
23: cleanup
24:
25: procedure popStack{Remove stale nodes in a stack and bookkeep the information about pattern matching}
26: input: a stack S_x
27: input: an integer p
28: while !empty(S_x) and top(S_x).end < p do
29:   m = top(S)
30:   if isComplete(m) then
31:     if isRoot(x) then
32:       output(m) {Output the candidates}
33:     else {x is not root of Q}
34:       for all elements m' in S_parent(x) do
35:         if desc-axis(x) or (child-axis(x) and m'.depth− m.depth = 1 ) then
36:           set(m', asChild(x,parent(x)))
37:           copy( m', m ){Move candidates to the nodes in the parent stack and continue to look for pattern match}
38:         end if
39:       end for
40:     end if
41:   end if
42:   clear(m)
43:   pop(S)
44: end while
```

As we can see, the elements in a stack can be represented as a vector $< start, end, depth, array\ of\ child\ match, candidates >$. The first three components are used to compare the structure relationship between XML nodes. The $array\ of\ child\ match$ and $candidates$ are required to process the predicates of a twig query correctly.

### *3.3.2* Algorithm

Algorithm 2 shows twig query processing of EXPedite.

The query processing over an XML stream proceeds as follows: When we process a node $n$ in an XML stream $F$, if it matches a stack $S_x$ (i.e. its tag matches query node $x$'s tag, line 13) and is a "qualified" node (line 14), we push it onto $S_x$. A node $n$ matching $S_x$ is *qualified* if either its depth matches the axis of the root stack, or an element in the parent stack $S_{parent(x)}$ and $n$ satisfy the axis relationship between $x$ and $parent(x)$ in $Q$. As we advance the XML stream to meet $n$, we pop out stale elements from stacks according to the position of $n$ (lines 10-12).

Before we pop an element, we must determine whether the candidates associated with it should be kept buffered (or output) or discarded and also perform some bookkeeping on elements in the parent stack. When an element $m$ in stack $S_x$ that is being popped out has found matches to all of its children in the twig $Q$, we report to elements in the parent stack which satisfy the axis in the query that $m$ is complete (line 36). Furthermore, if $m$ has candidates associated with it, we copy them to the elements in the parent stack (line 37). Then we clear the candidates of $m$ and pop out stack (line 42-43). When we pop out an element in the root stack, if all of its children in the query have been found, we are sure that a match of the whole query has been found so we output its candidates as part of query solutions (line 32).

Let us look at an example to see how the algorithm works.

**Example 3.1:** Consider a query $Q = //section[//figure/name]/title$ on the XML data in figure 1. To be concise, we re-draw the XML data in figure 2(a), excluding value information. $Q$ is represented as a twig in figure 2(b). For clarity, rather than using encoded nodes we use subscripted tag names to represent nodes in the XML stream and on stacks. Figure 2(c) represents the state of the stacks when $section_2$ is met. To avoid clutter, we use the initial letter to denote a tag for elements in stacks. Since the element $title_1$ in stack $S_{title}$ becomes stale, it should be popped out. Before we pop it, since $section_1$ in the parent stack and $title_1$ satisfy the child axis, we report to the element $section_1$ that a match for its child $title$ in $Q$ has been found (line 36). We put $title$ in parenthesis along with $section_1$ for illustration, and buffer $title_1$ as a candidate, pointed to by $section_1$ (line 37). Now the status of this candidate depends on whether or not $section_1$ can find matches on all of its children.

A snapshot of the stacks when $figure_1$ is met is shown in figure 2(d). As we can see, $title_2$ is reported to element $section_2$, buffered as a candidate of $section_2$ and finally popped out. Notice that $title_1$ and $title_2$ are pointed to by $section_1$ and $section_2$, respectively. This means that whether $title_1$ ($title_2$) should be output or discarded depends on whether $section_1$ ($section_2$) can find matches for all its children, i.e. whether it is complete. However the completeness of $section_1$ may be independent of that of $section_2$, so these two candidates should not be merged. The node $figure_1$ is pushed onto stack $S_{figure}$.

When $F$ is finished, all the elements in the stacks become stale. We pop the stacks in bottom up traversal order of the query tree. In this example, we first report $name_1$ to elements in parent stack: $figure_1$ (line 36) and pop it out (line 43); the snapshot is shown in figure 2(e). When we are about to pop $figure_1$, since both $section_1$ and $section_2$ satisfy the descendant axis with $figure_1$, we record for both elements that a match for the $figure$ child has found. At the time that we pop $section_2$ and $section_1$, since both query children have found matches, candidates $title_1$ and $title_2$ are output as query results.

■

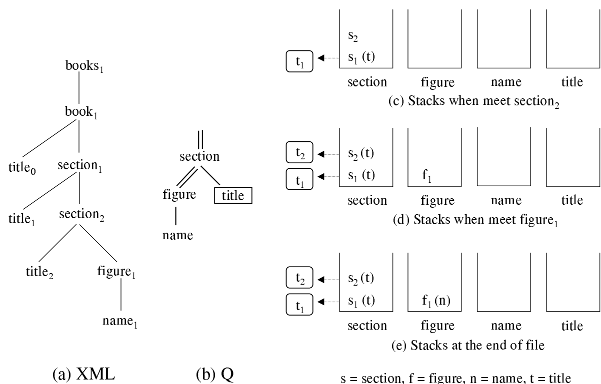



**Figure 2: Example of twig query processing**

**Theorem 3.2:** Algorithm $TwigQuery$ is correct. ■

The proof is based on induction over the height of the query twig pattern starting from the leaves. It is omitted here for reasons of space.

**Theorem 3.3:** The time complexity of algorithm $TwigQuery$ is $O(|F||Q|(|Q| + DB))$, where $|Q|$ and $|F|$ are the sizes of the XPath query and the XML stream respectively, $D$ is maximum number of repeated tags in a root-to-leaf path in the XML tree represented by $F$, and $B$ is the maximum size of the candidates. ■

PROOF. For each encoded XML node, it matches at most $|Q|$ query nodes (line 13). For each stack of the matching query node, an XML node will be pushed and poped at most once. The cost of checking whether a node need to be pushed and pushing a node to a stack is $O(|Q|)$ (line 14-15). The complexity of poping out a node is bounded by the cost of line 34-39. The loop is bounded by $D$, and candidates copy is bounded by $B$. The size of candidates $B$ is bounded by the XML stream size $F$, but in practice it is much smaller. Therefore the total cost of popping out a node is $DB$. The overall complexity is bounded by $O(|F||Q|(|Q| + DB))$. □

# 4. EXPERIMENTAL EVALUATION

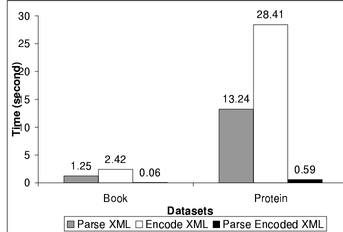
We have implemented EXPedite in C++ using the `icc` compiler. The XML parser used is the Xerces SAX2Parser. In this section, we present results of a detailed performance study of this implementation.

## 4.1 Experiment Setup

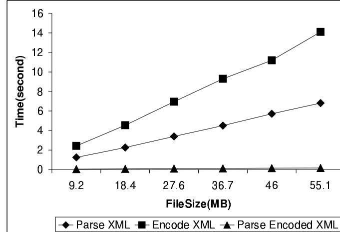
All experiments were conducted on a Pentium 4 1.5GHz machine with 512MB memory, running the Redhat 9 distribution of GNU/ Linux (kernel 2.4.20-8).

| Name | Streaming | return node is not leaf | "//" in predicate | path in predicate | multiple predicates for one query node | nested predicates | Buffered candidates |
|---|---|---|---|---|---|---|---|
| EXPedite | X | X | X | X | X | X | X |
| Path/TwigStack | | X | X | X | X | X | X |
| XSQ | X | X | | | | | X |
| XMLTK | X | | | | X | | |

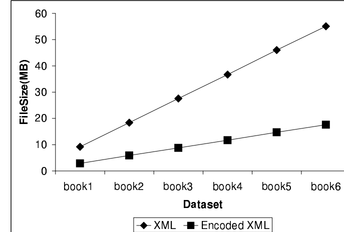
**Figure 3: Features of several systems**



(a) Parsing/Encoding Time for Different Datasets



(b) Parsing/Encoding Time as File Size Increases



(c) XML and Encoded XML File Size

**Figure 4: Performance for XML Encoding**

We compare EXPedite with several systems which support XPath query processing: *Path/TwigStack* [5], *XSQ* [24] and *XMLTK* [14]. Figure 3 summarizes some basic features of these systems.

We conducted experiments on two datasets. The first dataset is a real dataset from the International Protein Sequence Database [12]. The DTD of the protein file 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 [16], which takes a DTD as input, and produces documents that conform to that DTD, according to a set of workload parameters. We use the `Book` DTD from the XQuery use cases [28] as the input DTD. The `Book` DTD is cyclic, and is used to generate XML documents that contain multiple levels of recursion. There are only 12 tags in the DTD, and the generated data values are small. We use the default settings of XML Generator for all the parameters except for the following two: `DocDepth` and `MaxRepeats`. `DocDepth` bounds the maximum depth of the XML document generated. In this work, we are less concerned with the absolute document depth than with the depth of recursive elements. This is because when "//" axes are present in queries, a recursive element can participate in more than one query pattern match, which significantly impacts the efficiency of query processing. We specify DocDepth to be 20. `MaxRepeats` determines the maximum number of times an element can repeat in its parent element, and is set to 9.

## 4.2 Performance of XML Encoding

Figure 4(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. This proves that the compute power required for parsing encoded XML data is very small, hence suitable for constrained devices.

Figure 4(b) shows that the time for parsing XML, encoding XML and parsing encoded XML data increases linearly with the XML file size. The cost of parsing encoded data increases by very little.

Figure 4(c) shows that the encoded XML data is much smaller than the original XML file, a side effect of EXPedite which is very beneficial in the data exchange scenario.

`Book` Dataset

| | |
|---|---|
| $Q_1$ | /book//section//title |
| $Q_2$ | /book//section/figure/image/@source |
| $Q_3$ | /book//section[//title]/figure |
| $Q_4$ | /book//section/figure/image[@source = CDATA24553] |
| $Q_5$ | //section[figure/image/@source]title |
| $Q_6$ | //figure[@height, @width] |

`Protein` Dataset

| | |
|---|---|
| $Q_1$ | /PD//protein/name |
| $Q_2$ | /PD//PE//reference/refinfo//xrefs//xref//db |
| $Q_3$ | //PE/reference/refinfo[year=1988]/title |
| $Q_4$ | //PE//refinfo[volume]//author |
| $Q_5$ | //PE[protein/name]/organism |
| $Q_6$ | //PE[//refinfo[title, citation/@type]]/@id |

"PD" and "PE" are shorthand for "ProteinDatabase", and "ProteinEntry", respectively.

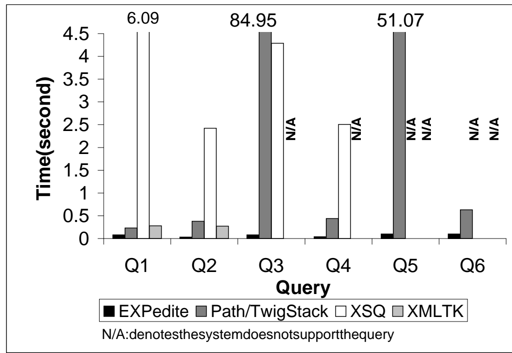
**Figure 6: Query sets**

## 4.3 Query Processing Time

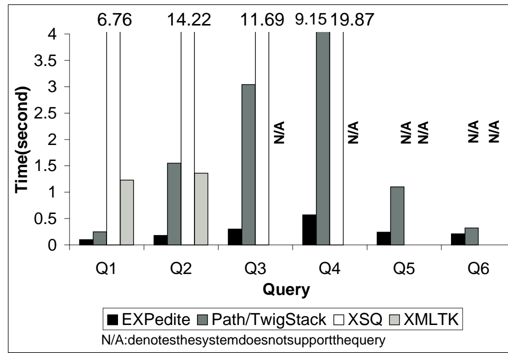
Next we compare XPath query processing on regular XML data with encoded XPath query processing on encoded XML data using EXPedite.

The queries we tested are listed in figure 6. $Q_1$ is a short path query, and $Q_2$ is a relatively long path query. $Q_3$ and $Q_4$ are two simple twig queries which only involve a single tag or attribute name in the branch. $Q_3$ has a value predicate and produces a small result. $Q_5$ and $Q_6$ are two complex twig queries, which involve multiple predicates for a single node, path expressions in a predicate, nested predicates, or descendant axes in predicates. As presented in figure 3, XMLTK only supports $Q_1$ and $Q_2$, XSQ supports $Q_1$, $Q_2$, $Q_3$ and $Q_4$, while Path/TwigStack and EXPedite support all the queries.

EXPedite, XMLTK and Path/TwigStack are implemented in C++, while XSQ is implemented in Java. In order to get




(a) `Book` dataset (9 MB)




(b) `Protein` dataset (75 MB)

**Figure 5: Query Execution Time**

a fair comparison, we referenced a Java performance benchmark against C [6] which tested 8 different benchmarks for various Java execution environments. We found the closest execution environment to be the Sun JDK 1.3.0 client versus the C++ compiler gcc 2.91.66 running on a 700 MHz Pentium III Linux machine. We took the ratio of the `Series` benchmark in which Java had the worst performance compared to C (1: 0.484321) and normalized the execution time of XSQ in the following comparison.

The Path/TwigStack algorithm [5] is designed for an XML database. It returns the node ids as the query result rather than the actual XML fragments. Since the implementation that we were able to obtain of Path/TwigStack does not support value predicates, we evaluate $Q_3$ for both datasets by removing the value predicates. The comparison between it and other systems as presented in this paper is therefore only an approximation.

As shown in section 4.2, parsing encoded XML data is much more efficient than parsing regular XML data using a standard SAX Parser. In this section, we exclude the parsing time from the query processing time in order to compare the query processing performance. 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(a) reports the query execution time of different XML query processing systems for the `Book` data over different queries. Since TwigStack processes a twig query by decomposing it to subqueries and merge-joining the intermediate results at the end, the query performance degrades when the join is expensive and/or the intermediate results are large. For example, $Q_3$ and $Q_5$ involve a three-way join of the intermediate results of the subqueries, and are expensive compared to $Q_2$ and $Q_4$. Since $Q_3$ involves descendant axis in the predicate, it is not supported by XSQ. We make an approximate comparison by changing it to $/book//section[/title]/figure$ and show the execution time in figure 5(a).

Figure 5(b) reports the query execution time of different XML query processing systems for the `Protein` data over different queries. Since the DTD of the `Protein` dataset is a tree, the difference in performance of different systems is not as dramatic as the `Book` dataset, whose DTD is recursive.

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 7 reports the query processing time for increasing sizes of XML data on $Q_1$, $Q_2$, $Q_3$, $Q_4$, $Q_5$ and $Q_6$, respectively. As we can see, as the file size increases, the execution time of EXPedite increases much more slowly than other approaches.

## 4.5 Query Response Time

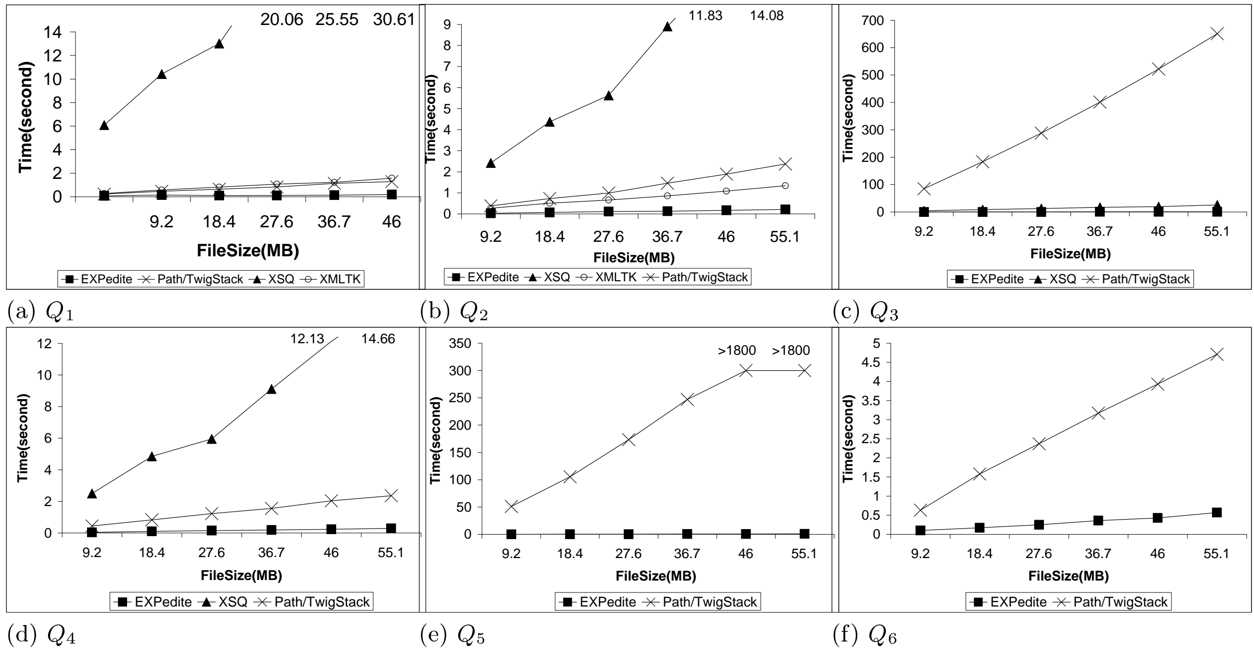
Response time is critical in a stream processing environment since fast response allows users to get real time information. We measured the average response time over all query results produced, for every query on both datasets. We excluded half of the parsing time in the measurements. From figure 8, we can see that the response time of EXPedite is fast since it outputs each candidate query result as soon as it can be verified to be correct.

# 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 XML labels $< start, end, level >$ [19, 30, 15, 10, 3, 5, 8, 17, 29]. [19, 30, 15, 10, 3] focus on how to efficiently evaluate queries containing one axis when the XML data is stored in a relational database. [8, 17, 29] explore the same problem and design specialized indexes to improve performance: augmented B+-trees with sibling pointers, *XR-Tree* and *PBi-Tree*, respectively. [5] proposes the *PathStack* algorithm to evaluate a path query as a whole. Furthermore, a twig query is answered by decomposing it to path queries and joining the intermediate results of path pattern matches. EXPedite also takes advantage of XML node labeling. It processes twig queries in a streaming fashion, without decomposing them and generating intermediate results.

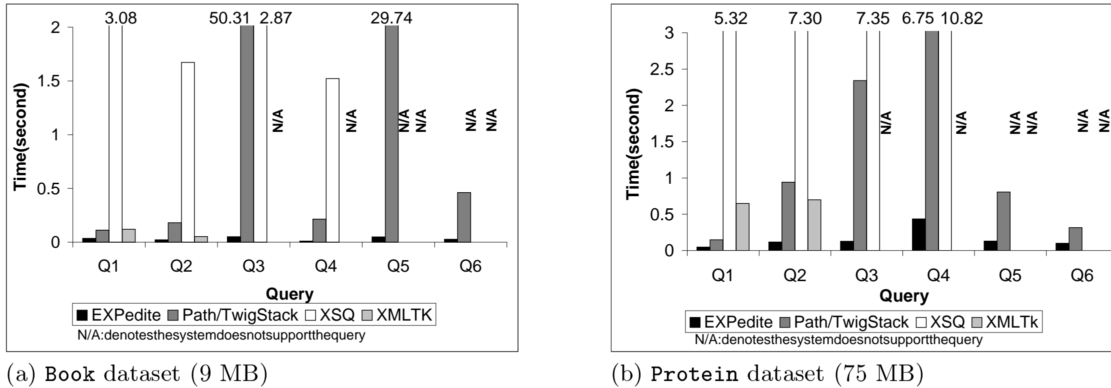
Another class of XML query processing is performed in a streaming fashion. *XSQ* [24] is the most recent work on XPath query processing containing /, // and [] on streaming XML data. It uses a hierarchical pushdown transducer




(a) $Q_1$ (b) $Q_2$ (c) $Q_3$

(d) $Q_4$ (e) $Q_5$ (f) $Q_6$

**Figure 7: Query Execution Time as File Size Increases**




(a) `Book` dataset (9 MB) (b) `Protein` dataset (75 MB)

**Figure 8: Query Response Time**

(HPDT) to model query processing. At each location step, at most one predicate is supported, and within a predicate, either one child axis or one attribute with an optional value comparison is allowed. A transducer-based approach, the XML streaming machine *XSM*, is presented in [21] to answer XQueries without descendant axes. [23] presents *SPEX*, a streamed evaluation of XPath containing /, //,* and [] against XML streams by a network of transducers. At most one predicate is allowed at each location step. [4] supports path query processing containing /, // and * on XML streams using deterministic finite automata (DFA).

A closely related question is how to filter a large number of XML queries efficiently in a publish-subscribe system. [14] approaches XML query processing by constructing DFAs in a lazy way, with new states added as needed at runtime so that the number of states in the lazy DFA is manageable. *YFilter* [11] proposes to use a nondeterministic finite state automaton-based (NFA) representation of path expressions to enable efficient shared processing for common expressions on different XPath queries. Methods for indexing common subexpressions of XPath queries which contain child axes using a data structure called *XTrie* are presented in [7]. The *MatchMake* [18] system uses a "requirement index", the dual to a traditional index, to enable efficient XML matching to a twig query. The algorithms require at most two passes over an input XML document.

For a general discussion of XML query processing in a non-streaming fashion, we refer the reader to a number of recent papers or products on that topic, for example, Galax [26] and [13].

There are many works on XML Binary format proposed in the W3C Workshop on Binary Interchange of XML Information Item Sets [25], and XML data compression, including XMill [20], XGrind [27] and XPress [22], where the goal is to save parsing time and transmission bandwidth. EXPedite focuses on speeding up query processing on XML data without decoding, and can have compression applied to the encoded data to further reduce the size during transmission if necessary.

# 6. CONCLUSIONS

This paper presents the EXPedite system: a novel model of data processing in an information exchange environment. We propose a general and effective encoding scheme of XML data for data-sending servers, and design algorithms for data receivers to efficiently process twig queries over broadcast encoded XML stream. The proposed query processing algorithm based on an XML labeling scheme achieves an efficient query processing performance over encoded XML streams. By encoding, EXPedite "migrates" the power of the server to the receivers to enable efficient processing over the broadcast data. To the best of our knowledge, EXPedite is the first system to propose encoding XML data in order to achieve efficient query processing on constrained devices in a streaming fashion. Experiments demonstrate 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 twig queries. In the future we will extend the encoder to fully support XML 1.0 and extend the query processor to handle other features in the XPath language. We will also exploit other XML encoding schemes and study how they affect the query processing algorithm.

# 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] I. Avila-Campillo, T. J. Green, A. Gupta, M. Onizuka, D. Raven, and D. Suciu. XMLTK: An XML Toolkit for Scalable XML Stream Processing. In *Programming Language Technologies for XML(PLAN-X)*, 2002.

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

[6] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for scientific applications. In *Java Grande*, pages 97–105, 2001.

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

[8] 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.

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

[10] D. DeHaan, D. Toman, M. Consens, and M. T. Ozsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proceedings of SIGMOD*, 2003.

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

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

[13] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proceedings of PODS*, 2003.

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

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

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

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

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

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

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

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

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

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

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

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

[26] J. Simon and M. Fernndez. Galax. http://db.bell-labs.com/galax.

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

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

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

[30] 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.