Indexing and Querying XML Data for Regular Path Expressions [*]

Quanzhong Li *Bongki Moon*

Dept. of Computer Science
University of Arizona, Tucson, AZ 85721
{lqz,bkmoon}@cs.arizona.edu

Abstract

With the advent of XML as a standard for data representation and exchange on the Internet, storing and querying XML data becomes more and more important. Several XML query languages have been proposed, and the common feature of the languages is the use of regular path expressions to query XML data. This poses a new challenge concerning indexing and searching XML data, because conventional approaches based on tree traversals may not meet the processing requirements under heavy access requests. In this paper, we propose a new system for indexing and storing XML data based on a numbering scheme for elements. This numbering scheme quickly determines the ancestor-descendant relationship between elements in the hierarchy of XML data. We also propose several algorithms for processing regular path expressions, namely, (1) $\mathcal{EE}$**-Join** for searching paths from an element to another, (2) $\mathcal{EA}$**-Join** for scanning sorted elements and attributes to find element-attribute pairs, and (3) $\mathcal{KC}$**-Join** for finding Kleene-Closure on repeated paths or elements. The $\mathcal{EE}$**-Join** algorithm is highly effective particularly for searching paths that are very long or whose lengths are unknown. Experimental results from our prototype system implementation show that the proposed algorithms can process XML queries with regular path expressions by up to an order of magnitude faster than conventional approaches.

[*] This work was sponsored in part by National Science Foundation CAREER Award (IIS-9876037) and Research Infrastructure program EIA-0080123. The authors assume all responsibility for the contents of the paper.

Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001

1 Introduction

The extensible markup language XML has recently emerged as a new standard for information representation and exchange on the Internet [4]. Since XML data is self-describing, XML is considered one of the most promising means to define semi-structured data, which is expected to be ubiquitous in large volumes from diverse data sources and applications on the web. XML allows users to make up any new tags for descriptive markup for their own applications. Such user-defined tags on data elements can identify the semantics of data. The relationships between elements can be defined by nested structures and references. For example, XML metadata can be used to describe a web site structure to facilitate navigation by generating interactive site maps [19].

To retrieve XML and semi-structured data, several query languages have been proposed. Examples are Lorel [1], XML-QL [10], XML-GL [5], Quilt [7], XPath [8], and XQuery [6]. The XQuery is the first public working draft of a query language for XML released recently from the World Wide Web Consortium (W3C). The XQuery language is designed to be broadly applicable across all types of XML data sources from documents to databases and object repositories. The common features of these languages are the use of regular path expressions and the ability to extract information about the schema from the data [14]. Users are allowed to navigate through arbitrary long paths in the data by regular path expressions. For example, XPath uses path notations as in URLs for navigating through the hierarchical structure of an XML document.

Despite the past research efforts, it is widely believed that the current state of the art of the relational database technology fails to deliver all necessary functionalities to efficiently store XML and semi-structured data. Furthermore, when it comes to processing regular path expression queries, only a few straightforward approaches based on conventional tree traversals have been reported in the literature (*e.g.*, [20]). Such approaches can be fairly inefficient for processing regular path expression queries, because the overhead of traversing the hierarchy of XML data can be substantial if the path lengths are very long or unknown.

In this paper, we propose a new system called **XISS** for indexing and storing XML data based on a new numbering scheme for elements and attributes. The index structures of **XISS** allow us to efficiently find all elements or attributes with the same name string, which is one of the most common operations to process regular path expression queries. The proposed numbering scheme quickly determines the ancestor-descendant relationship between elements and/or attributes in the hierarchy of XML data. We also propose several algorithms for processing regular path expression queries, namely, (1) $\mathcal{EE}$-**Join** for searching paths from an element to another, (2) $\mathcal{EA}$-**Join** for scanning sorted elements and attributes to find element-attribute pairs, and (3) $\mathcal{KC}$-**Join** for finding Kleene-Closure on repeated paths or elements. The $\mathcal{EE}$-**Join** algorithm is highly effective particularly for searching paths that are very long or whose lengths are unknown.

Main contributions of the proposed solutions are:

- The proposed numbering scheme is designed based on the notion of *extended preorder* to accommodate future insertions gracefully. This numbering scheme allows us to determine the ancestor-descendant relationship between elements and attributes in constant time.
- Three major index structures are proposed, namely, *element index*, *attribute index* and *structure index*. They are used to index and store XML data objects, and support search by both name string and structure efficiently.
- The proposed join algorithms can process regular path expression queries without traversing the hierarchy of XML data. Experimental results from our prototype system implementation show that the proposed algorithms can process XML queries up to 10 times faster than conventional approaches.

The rest of this paper is organized as follows. In Section 2, we present the numbering scheme and major index structures of the proposed **XISS** system. In Section 3, we discuss the potential inefficiency of conventional query processing for regular path expressions. Then, we describe the decomposition of regular path expressions and present the proposed join algorithms. Section 4 presents the results of experimental evaluation of the proposed **XISS** system and the join algorithms. We overview related work briefly in Section 5. Finally, Section 6 summaries the contribution of this paper and gives outlook to future work.

2 XISS: XML Indexing and Storage System

XML data can be queried by a combination of value search and structure search. Search by value can be done by matching such XML values as document names, element names/values, and attribute names/values. Search by structure can be done mostly by examining ancestor-descendant relationships given in regular path expression queries. To facilitate XML query processing by both value and structure searches, it is crucial to provide mechanisms to quickly determine the ancestor-descendant relationship between XML elements as well as fast accesses to XML values.

In this section, we propose a numbering scheme for XML documents, elements and attributes, which enables efficient search by value and structure. We then propose a new XML Indexing and Storage System (**XISS**) composed of three major index structures (namely, *element index*, *attribute index*, and *structure index*), a data loader and a query processor.

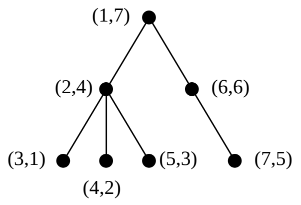
2.1 Numbering Scheme

XML data objects are commonly modeled by a tree structure, where nodes represent elements, attributes and text data, and parent-child node pairs represent nesting between XML data components. To speed up the processing of regular path expression queries, it is important to be able to quickly determine ancestor-descendant relationship between any pair of nodes in the hierarchy of XML data. For example, a query with a regular path expression "`chapter3/_*/figure`" is to find all `figure` elements that are included in `chapter3` elements. Once all `chapter3` elements and `figure` elements are found, those two element sets can be *join*ed to produce all qualified `chapter3`-`figure` element pairs. This join operation can be carried out without traversing XML data trees, if the ancestor-descendant relationship for a pair of `chapter3` and `figure` elements can be determined quickly. This is the main idea of the proposed algorithms, which will be presented in Section 3.

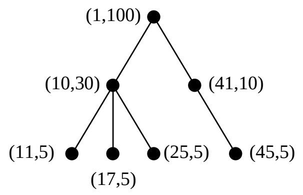
To the best of our knowledge, it was Dietz's numbering scheme that was the first to use tree traversal order to determine the ancestor-descendant relationship between any pair of tree nodes [12]. His proposition was: *for two given nodes x and y of a tree T, x is an ancestor of y if and only if x occurs before y in the preorder traversal of T and after y in the postorder traversal.* For example, consider a tree in Figure 1(a) whose nodes are annotated by Dietz's numbering scheme. Each node is labeled with a pair of preorder and postorder numbers. In the tree, we can tell node (1,7) is an ancestor of node (4,2), because node (1,7) comes before node (4,2) in the preorder (*i.e.,* $1 < 4$) and after node (4,2) in the postorder (*i.e.,* $7 > 2$).

An obvious benefit from this approach is that the ancestor-descendant relationship can be determined in constant time by examining the preorder and postorder numbers of tree nodes. On the other hand, the limitation of this approach is the lack of flexibility. That is, the preorder and postorder may need to be recomputed for many tree nodes, when a new node is inserted. To get around this problem, we propose a new numbering scheme that uses an *extended preorder* and a *range of descendants*. The proposed numbering scheme associates each node with a pair of numbers $<order, size>$ as follows.

- For a tree node y and its parent x, $order(x) < order(y)$ and $order(y) + size(y) \leq order(x) + size(x)$. In other words, interval $[order(y), order(y) + size(y)]$ is contained in interval $[order(x), order(x) + size(x)]$.
- For two sibling nodes x and y, if x is the predecessor of y in preorder traversal, $order(x) + size(x) <$



(a) Dietz's Numbering Scheme using Preorder and Postorder



(b) Proposed Numbering Scheme using $<order, size>$ pair

Figure 1: Numbering Scheme Examples

$order(y)$.

Then, For a tree node x, $size(x) \geq \sum_y size(y)$ for all y's that are a direct child of x. Thus, $size(x)$ can be an arbitrary integer larger than the total number of the current descendants of x, which allows to accommodate future insertions gracefully.

It is not difficult to show that the nodes ordered by this proposed numbering scheme is equivalent to that of preorder traversal. That is, the proposed numbering scheme guarantees that for a pair of tree nodes x and y, $order(x) < order(y)$ if and only if x comes before y in preorder traversal. Furthermore, the ancestor-descendant relationship for a pair of nodes can be determined by examining their $order$ and $size$ values. In Figure 1(b), each node is labeled by a $<order, size>$ pair, which defines an interval. The interval of a node is properly contained in the interval of its parent node. For example, a node (25,5) is contained in both (10,30) and (1,100). Hence, the node with order 25 is a descendant of nodes with order 10 and 1. This observation leads to the following lemma.

Lemma 1 *For two given nodes x and y of a tree T, x is an ancestor of y if and only if $order(x) < order(y) \leq order(x) + size(x)$.*

Proof. Proof by induction. □

Compared with Dietz's scheme, our numbering scheme is more flexible and can deal with dynamic updates of XML data more efficiently. Since extra spaces can be reserved in what we call extended preorder to accommodate future insertions, global reordering is not necessary until all the reserved spaces (*i.e.*, unused order values) are consumed. Note that for both numbering schemes, deleting a node does not cause renumbering the nodes. However, it is easier for our numbering scheme to *recycle* the order values of deleted nodes.

Both elements and attributes use the $order$ of the $<order, size>$ pair as their unique identifier in the document tree. As for attributes, additional care needs to be exercised to ensure that attribute nodes are placed before their sibling elements in the order by the numbering scheme. We will discuss later in Section 3.3 how this can enable faster element and attribute join operations.

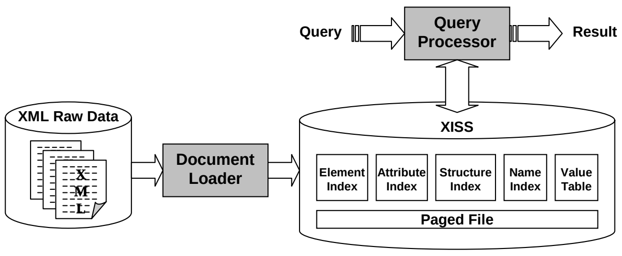


Figure 2: Index Structure Overview

2.2 Index and Data Organization

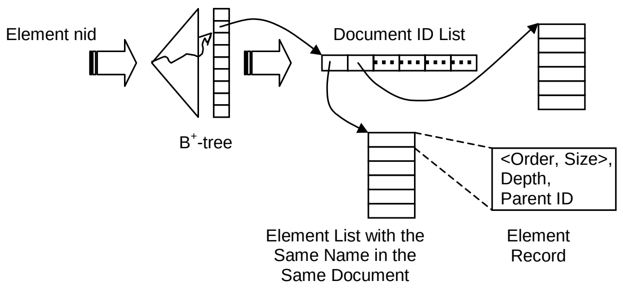
As is mentioned above, the **XISS** system supports search by element or attribute name and structure. To achieve this goal, the **XISS** system provides mechanisms to process the following operations efficiently.

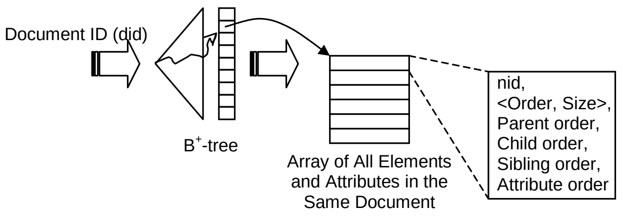
- For a given element name string, say `figure`, find a list of elements having the same name string (*i.e.*, `figure`), grouped by documents which they belong to.
- For a given attribute name string, say `caption`, find a list of attributes having the same name string (*i.e.*, `caption`), grouped by documents which they belong to.
- For a given element, find its parent element and child elements (or attributes). For a given attribute, find its parent element.

The index structure of **XISS** is composed of three major components: *element index*, *attribute index* and *structure index*, which are shown in Figure 2. The other two components in Figure 2 are *name index* for storing name strings and *value table* for values.

Since all value entities in XML data are considered variable-length character strings, all distinct name strings are collected in the *name index*, which is implemented as a B$^+$-tree. Then, each distinct name string is uniquely identified by a *name identifier* (or *nid*) returned from the name index. The use of name index minimizes storage and computational overhead by eliminating replicated strings and string comparisons. For the same reason, all string values (i.e. attribute value and text value) are collected in *value table*. Each XML document is also assigned a unique document identifier (did), which is an index key to retrieve the document name. An element or attribute can be uniquely identified by its *order* and *did* in the entire system.

The *element index*, *attribute index* and *structure index* are the three indexes to support the three essential functionalities listed above, respectively. Both the element index and attribute index are implemented as a B$^+$-tree using name identifiers (nid) as keys. Each entry in a leaf node points to a set of fixed-length records for elements (or attributes) having an identical name string, grouped by document they belong to. The element index allows us to quickly find all elements with the same name string,


Element nid
B+-tree
Document ID List
<Order, Size>, Depth, Parent ID
Element List with the Same Name in the Same Document
Element Record
Figure 3: Element Index


Document ID (did)
B+-tree
Array of All Elements and Attributes in the Same Document
nid, <Order, Size>, Parent order, Child order, Sibling order, Attribute order
Figure 4: Structure Index

which is one of the most common operations to process regular path expression queries. Each element record includes an $<order, size>$ pair and other related information of the element, and the element records are in a sorted order by the $order$ values as shown in Figure 3.

The attribute index has almost the same structure as the element index, except that the record in attribute index has a value identifier *vid*, which is a key used to obtain the attribute value from the value table.

The organization of the structure index is shown in Figure 4. It is a collection of linear arrays, each of which stores a set of fixed-length records for all elements and attributes from an XML document. Within an array, the elements and attributes are together sorted by their $order$ value (*i.e.*, in preorder traversal). Each record of the structure index stores a name identifier (nid), $order$ values of the first sibling, first child, and the first attribute and so on.

3 Path Join Algorithms

In this section, we propose new path-join algorithms to efficiently process regular path expression queries for XML data. Consider the following sample query borrowed from the XQuery working draft [6].

```
(Q1):/chapter/_*/figure[@caption="Tree Frogs"]
```

Symbol	Function of Symbol
_	Denotes any single node
/	Denotes a separator between nodes in a path
|	Denotes a union of nodes
?	Denotes zero or one occurrence of a node
+	Denotes one or more occurrences of a node
*	Denotes zero or more occurrences of a node
[]	Encloses a predicate expression
@	Denotes attributes
()	Indicates precedence

Table 1: Notations for Regular Path Expressions

The query `Q1` is to find all figures with a caption `Tree Frogs` in all chapters. In this query, `chapter` and `figure` are XML elements, and `caption` is an XML attribute. This query will be used as a running example in the following sections. Note that the notations used in this paper are slightly different from those used in XQuery working draft. See Table 1 for the full notations for regular path expressions that we use in this paper.

3.1 Conventional Approaches

Most straightforward approaches to processing regular path expression queries like `Q1` is to traverse the hierarchy of XML objects in either *top-down* or *bottom-up* fashion [20]. To process the query `Q1` by a top-down approach, for example, all downward paths starting from a `chapter` element should be followed to find out whether there exists any `figure` element as a descendant. This step needs to be repeated for all `chapter` elements in XML database. This implies that it is absolutely necessary to examine every possible path from each `chapter` elements to all leaf nodes in XML trees, because it is not usually known where `figure` elements will be found in the paths. If a `chapter` element is the root of an XML tree, then the entire tree will be traversed.

The cost of tree traversal may be reduced by a bottom-up approach. For the same query `Q1`, all `figure` elements with a caption `Tree Frogs` will be searched. Then, from each of such `figure` elements, a corresponding XML tree will be examined by traversing up the tree to find out whether there exists any `chapter` element as an ancestor. This upward traversal will be simpler and less costly, because there exists always at most one upward path. However, if there are many `figure` elements with a caption `Tree Frogs` and only a few `chapter` elements, the cost of bottom-up approach might be even higher than that of top-down approach.

A hybrid approach has been proposed that traverses in both top-down and bottom-up fashions, meeting in the middle of a path expression [20]. This hybrid approach can take advantages of top-down and bottom-up approaches for XML data of certain structural characteristics. However, its effectiveness is not always guaranteed. In the following sections, we describe the decomposition of a regular path expression, and propose new path-join algorithms to process regular path expression queries without traversing XML trees.

3.2 Decomposition of Path Expressions

The main idea of the proposed path-join algorithms is that a complex path expression is decomposed into several simple path expressions. Each simple expression produces an intermediate result that can be used in the subsequent stage of processing. The results of the simple path expressions are then combined or joined together to obtain the final result of a given query. There is an interesting analogy between the way a regular path expression is decomposed, processed and combined, and the way a multi-way join operation is processed in a series of two-way joins by a relational query processor. For example, a regular path expression of the form $\texttt{E}_1\texttt{/E}_2\texttt{/E}_3\texttt{/E}_4$ with four elements $\texttt{E}_1$ through $\texttt{E}_4$ can be decomposed to

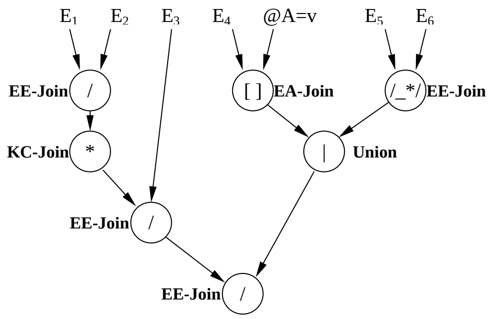


Figure 5: Decomposition of a Path Expression `Q2`

E_1/E_2 and E_3/E_4. Then, the intermediate results from E_1/E_2 and E_3/E_4 are joined together.

In general, a regular path expression can be decomposed to a combination of the following basic subexpressions:

1. a subexpression with a single element or a single attribute,
2. a subexpression with an element and an attribute (*e.g.*, `figure[@caption = "Tree Frogs"]`),
3. a subexpression with two elements (*e.g.*, `chapter/figure` or `chapter/_*/figure`),
4. a subexpression that is a Kleene closure (`+`,`*`) of another subexpression, and
5. a subexpression that is a union of two other subexpressions.

Figure 5 illustrates a way of decomposing a complicated regular path expression `Q2`.

(Q2): $(E_1/E_2)^*/E_3/((E_4[@A=v])|(E_5/\_^*/E_6))$

The leaf nodes at the top of the figure are subexpressions with a single element or attribute. Each circle represents one of the other four subexpression types (2) through (5) described above.

A subexpression with a single element or attribute can be processed by accessing the element index or attribute index of the **XISS** system described in Section 2.2. A union of two subexpressions can be processed by merging two intermediate results and grouping by documents in a straightforward way. For the other three subexpression types (2), (3) and (4), we propose three path-join algorithms, namely,

- $\mathcal{EA}$**-Join** for a subexpression type (2),
- $\mathcal{EE}$**-Join** for a subexpression type (3), and
- $\mathcal{KC}$**-Join** for a subexpression type (4).

Each of these three algorithms will be described in the following sections.

3.3 $\mathcal{EA}$-Join Algorithm

The $\mathcal{EA}$**-Join** algorithm joins two intermediate results from subexpressions, which are a list of elements and a list of attributes. For example, a regular path expression `figure[@caption = "Tree Frogs"]` searches all `figure` elements with a caption `Tree Frogs` from all XML documents in the **XISS** system. The intermediate results as input to $\mathcal{EA}$**-Join** algorithm are a list of `figure` elements and a list of `caption` attributes grouped by documents which they belong to. The $\mathcal{EA}$**-Join** algorithm is described in Algorithm 1.

Algorithm 1: $\mathcal{EA}$**-Join**: Element and Attribute Join

```
Input:  {$E_1, \ldots, E_m$}: $E_i$ is a set of elements
            having a common document identifier;
        {$A_1, \ldots, A_n$}: $A_j$ is a set of attributes
            having a common document identifier;
Output: A set of $(e, a)$ pairs such that the element $e$
            is the parent of the attribute $a$.

// Sort-merge {$E_i$} and {$A_j$} by doc. identifier.
1: foreach $E_i$ and $A_j$ with the same did do
       // Sort-merge $E_i$ and $A_j$
       // by PARENT-CHILD relationship.
2:     foreach $e \in E_i$ and $a \in A_j$ do
3:         if ($e$ is a parent of $a$) then output $(e, a)$;
       end
   end
```

Since the element (or attribute) index maintains the element (or attribute) records in a sorted order by document identifiers and then $order$ values, the join of the intermediate results can be obtained by a *two-stage sort-merge* operation without additional cost of sorting. That is, element sets and attribute sets are merged by document identifiers in the first stage. Then, in the second stage, for a pair of element list and attribute list with a matching document identifier (*i.e.*, extracted from the same document), the elements and attributes are merged by examining the parent-child relationship based on their $order$ values by the numbering scheme.

As we mentioned briefly in Section 2.1, it is important to ensure that attributes are placed before their sibling elements in the order by the numbering scheme. Its performance impact on the $\mathcal{EA}$**-Join** operation is potentially very high, because this additional requirement on the numbering scheme guarantees that those elements and attributes with a matching document identifier can be merged in a *single scan*. Specifically, both the lists $\{E_i\}$ and $\{A_j\}$ grouped by document are scanned once by the outer **foreach** loop (line 1 in Algorithm 1), and both the element list E_i and attribute list A_j are scanned once by the inner **foreach** loop (line 2 in Algorithm 1).

This can be best explained by an example shown in Figure 6. Note that tree nodes are annotated by $<order, size>$ pairs. Consider the XML tree at the left hand side of Figure 6, where an attribute `name`$<4,0>$ is numbered *after* its sibling element `chapter`$<2,1>$. By the time the parent-child relationship between `chapter`$<1,3>$ and `name`$<4,0>$ is examined, the attribute `name`$<3,0>$ has already been passed over. Consequently, to examine the parent-child relationship between `chapter`$<2,1>$ and `name`$<3,0>$, the attribute lists must be rescanned. In contrast, in the XML tree at the right hand side, the attribute `name`$<2,0>$ is numbered *before* its sibling element `chapter`$<3,1>$. The parent-child relationships for `chapter`$<1,3>$ and `name`$<2,0>$ pair and `chapter`$<3,1>$ and `name`$<4,0>$ pair can be determined without rescans.

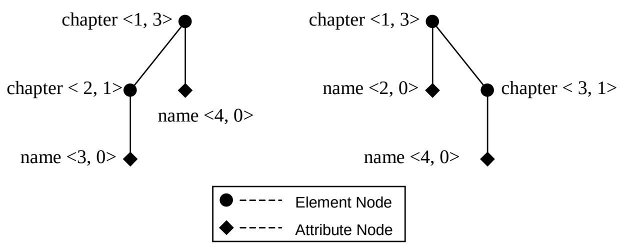


Figure 6: Examples of Correct and Incorrect Cases

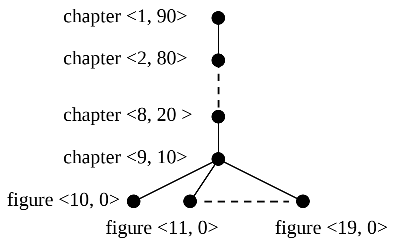


Figure 7: An Extreme Case of Element-Element Join

3.4 $\mathcal{EE}$-Join Algorithm

The $\mathcal{EE}$-**Join** algorithm joins two intermediate results, each of which is a list of elements obtained from a subexpression. For example, a regular path expression `chapter/_*/figure` searches all `chapter-figure` pairs that are in ancestor-descendant relationship from all XML documents in the **XISS** system. The intermediate results as input to $\mathcal{EE}$-**Join** algorithm are a list of `chapter` elements and a list of `figure` elements grouped by documents which they belong to. The $\mathcal{EE}$-**Join** algorithm is described in Algorithm 2.

Algorithm 2: $\mathcal{EE}$-**Join**: Element and Element Join

```
    Input:  {$E_1, \ldots, E_m$} and {$F_1, \ldots, F_n$}: $E_i$ or $F_j$
            is a set of elements having a common
            document identifier.
    Output: A set of $(e, f)$ pairs such that the element $e$
            is an ancestor of the element $f$.

    // Sort-merge {$E_i$} and {$F_j$} by doc. identifier.
1:  foreach $E_i$ and $F_j$ with the same did do
        // Sort-merge $E_i$ and $F_j$
        // by ANCESTOR-DESCENDANT relationship.
2:      foreach $e \in E_i$ and $f \in F_j$ do
3:          if ($e$ is an ancestor of $f$) then output $(e, f)$;
        end
    end
```

Like $\mathcal{EA}$-**Join** algorithm, $\mathcal{EE}$-**Join** algorithm can perform the join of two sets of elements by a *two-stage sort-merge* operation without additional cost of sorting. That is, both element sets are merged by document identifiers in the first stage. Then, in the second stage, for a pair of element sets with a matching document identifier (*i.e.*, extracted from the same document), both the element sets are merged by examining the ancestor-descendant relationship based on their $<order, size>$ values by the numbering scheme.

Unlike $\mathcal{EA}$-**Join** algorithm, however, two sets of elements with a matching document identifier can *not* be merged in a single scan by $\mathcal{EE}$-**Join** algorithm. By Lemma 1, for a pair of elements `chapter` and `figure` as an example, their ancestor-descendant relationship is determined by examining whether the $order(\texttt{figure})$ (*i.e.*, a point in extended-preorder) is contained in $[order(\texttt{chapter}), order(\texttt{chapter}) + size(\texttt{chapter})]$ (*i.e.*, a range in extended-preorder). The join of two sets of elements by ancestor-descendant relationship can be viewed as a join of a range set and a point set. Just as a point can be contained in more than a range, an element `figure` can be a descendant of more than a `chapter` element. See Figure 7 for an extreme case, where every `chapter` element must match every `figure` element. Thus, it may be necessary to scan the list of `figure` elements more than once.

Despite the fact that an element set may have to be scanned multiple times by the inner **foreach** loop (line 2 in Algorithm 2), $\mathcal{EE}$-**Join** algorithm is still highly effective, particularly for searching paths that are long or whose lengths are unknown. In Section 4, we compare $\mathcal{EE}$-**Join** algorithm with conventional approaches based on tree traversal. The effectiveness of $\mathcal{EE}$-**Join** is corroborated by the experimental results.

It is worth noting that $\mathcal{EE}$-**Join** algorithm can support an element-element join with a fixed-length path as in `chapter/_/_/figure`, which searches `chapter-figure` element pairs that are in great-grandparent relationship. Coupled with the depth of each element in an XML tree, the numbering scheme can determine the great-grandparent relationship in constant time. Another special case that can be processed by $\mathcal{EE}$-**Join** algorithm is a subexpression like `chapter/_*[@caption]`. Although this subexpression contains a pair of element and attribute, $\mathcal{EA}$-**Join** algorithm cannot process it in a single scan. Thus, this subexpression should be processed by $\mathcal{EE}$-**Join** algorithm.

3.5 $\mathcal{KC}$-Join Algorithm

Algorithm 3: Kleene Closure Algorithm

```
    Input:  {$E_1, \ldots, E_m$}, where $E_i$ is a group of
            elements from an XML document.
    Output: A Kleene closure of {$E_1, \ldots, E_m$}.

    // Apply $\mathcal{EE}$-Join algorithm repeatedly.
1:  set $i \leftarrow 1$;
2:  set $K_i^C \leftarrow \{E_1, \ldots, E_m\}$;
3:  repeat
4:      set $i \leftarrow i + 1$;
5:      set $K_i^C \leftarrow \mathcal{EE}$-Join($K_{i-1}^C, K_1^C$);
    until ($K_i^C$ is empty);
6:  output union of $K_1^C, K_2^C, \ldots, K_{i-1}^C$;
```

The $\mathcal{KC}$-**Join** algorithm processes a regular path expression that represents zero, one or more occurrences of a subexpression (*e.g.*, `chapter*` or `chapter+`). In each processing stage, $\mathcal{KC}$-**Join** algorithm applies $\mathcal{EE}$-**Join** to the result from the previous stage repeatedly until no more results can be produced. For example, to find `chapter*`, $\mathcal{KC}$-**Join** obtains `chapter/chapter` by self-joining a set of

Data Set	Size (Byte)	Files	Elements	Attributes
Shakespeare	7.9M	37	327K (22)	0 (0)
SIGMOD	3.5M	989	839K (47)	4775 (3)
NITF100	7.7M	100	63K (124)	263K (142)
NITF1	5.3M	1	38K (86)	171K (106)

Table 2: XML Data Set

`chapter` elements. In the next stage, it obtains `chapter/chapter/chapter` by joining the results from `chapter/chapter` and `chapter`. The final result is the union of results from all previous stages. The $\mathcal{KC}$-**Join** algorithm is described in Algorithm 3.

4 Experiment

We implemented the prototype of **XISS** to store the XML data and index. A primitive query interface is provided in C++. The Gnome XML parser was used to parse XML data [23]. We also used the GiST C++ library [17] for B^+-tree indexing. Query processing is directly implemented using the query interface.

Experiments were performed on a Sun Ultrasparc-II workstation running on Solaris 2.7. This workstation has 256 MBytes of memory and 20 GBytes of disk storage (Seagate ST320423A) with Ultra 10 EIDE interface. The disk is locally attached to the workstation and used to store XML data and index. We used the direct I/O feature of Solaris for all experiments to avoid operating system's cache effects.

4.1 Data Sets and Performance Metrics

We have chosen two data sets (Shakespeare, SIGMOD) from real-world applications and two synthetic data sets generated by an XML Generator from IBM [11]. These data sets are described in the following and the characteristics of those data sets are summarized in Table 2. In the last two columns, the two numbers in each entry represent the total number of elements (or attributes) and the number of distinct elements (or attributes), respectively.

Shakespeare's Plays: This data set is the Shakespeare's plays in XML format, which is marked up by Jon Bosak and available in [9].

SIGMOD Record: This data set is the XML version of ACM SIGMOD Record[1]. There are many small files containing on-line issues of SIGMOD Record.

NITF100 and NITF1: Using XML Generator and the NITF2.5 (News Industry Text Format) as the DTD, we generated two different versions of XML data sets: one data set stored in a single large document file (NITF1) and the other data set stored in 100 separate document files (NITF100).

4.2 Performance of Query Processing

In this section, we present the performance measurements and analyze the proposed algorithms mostly for element-element join and element-attribute join operations. The conventional top-down and bottom-up methods are compared with the proposed algorithms. Because the cost of output generation is the same regardless of algorithms applied, the output cost is not included in the measurements. We have not measured the performance of $\mathcal{KC}$-**Join** algorithm, because it is largely determined by the performance of $\mathcal{EE}$-**Join** algorithm.

4.2.1 $\mathcal{EE}$-Join Query

The queries we used for element-element join operations are of the form $\texttt{E}_A$`/_*/`$\texttt{E}_B$. For example, a query `chapter/_*/figure` is to find all `figure` elements that are descendants of `chapter` element. The actual queries used in the experiments are shown in Table 3.

With all these queries, we compared $\mathcal{EE}$-**Join** with a bottom-up method. A top-down method was not used, because it was expected to be outperformed by the bottom-up method for the data sets. The bottom-up method processes queries in the following steps. First, search all elements with name $\texttt{E}_B$. Second, starting from each element $\texttt{E}_B$, traverse up the tree to find $\texttt{E}_A$ elements. Third, if an element $\texttt{E}_A$ is found, output the path from $\texttt{E}_A$ to $\texttt{E}_B$.

Figure 8(a) and Figure 8(b) show the elapsed time for real-world data sets (Shakespeare and SIGMOD) and synthetic data sets (NITF100 and NITF1), respectively. The $\mathcal{EE}$-**Join** algorithm performs well even for a small number of buffer pages. The bottom-up method takes longer time to process the same query, especially for synthetic data, if the size of buffer pool is small. This is because the $\mathcal{EE}$-**Join** algorithm accesses the sorted elements from disk in a sequential manner, while the bottom-up method accesses elements from the structure index almost randomly. This results in a relatively low rate of page faults for $\mathcal{EE}$-**Join** algorithm, and a relatively high rate of page faults for the bottom-up method. The vertical lines in Figure 8(b) show the severely elongated processing times by the bottom-up method in the extreme case of using only one buffer page. Obviously, beyond the point where more than enough buffer pages are available, all performance measurements remain constant irrespective of the number of buffer pages.

From all the experiments done with both real-world and synthetic data sets, $\mathcal{EE}$-**Join** algorithm outperformed the bottom-up method by a wide margin. For real-world data sets, $\mathcal{EE}$-**Join** was *an order of magnitude faster* than the bottom-up method. For synthetic data sets, $\mathcal{EE}$-**Join** was *about 6 to 10 times faster* than the bottom-up method.

We measured IO time separately and observed the same trend in performance as in Figure 8. Disk IO was the dominant cost factor of query evaluations. In our experiments, 60 to 90 percent of total elapsed time was spent on disk accesses by $\mathcal{EE}$-**Join** algorithm.

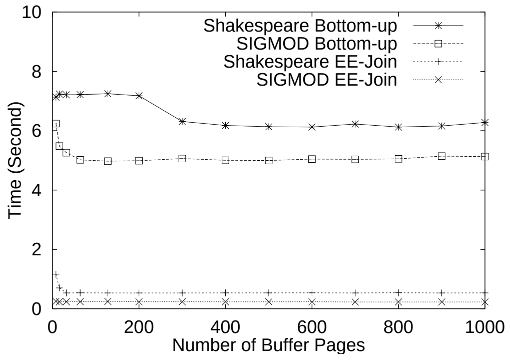
4.2.2 $\mathcal{EA}$-Join Query

The queries we used for element-attribute join operations are of the form `E[@A]`. For example, a query `figure[@caption]` is to find all `figure` elements with a `caption` attribute. For this type of queries, we compared the performance of $\mathcal{EA}$-**Join** algorithm with both

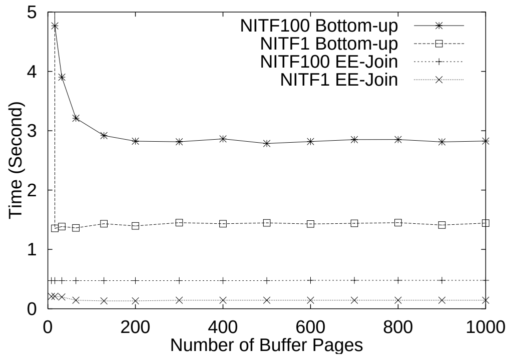
[1]http://www.acm.org/sigmod/record/xml

Data Set	Element E_A	Element E_B	# of E_A's	# of E_B's	# of Results
Shakespeare	ACT	SPEECH	185	31028	30951
SIGMOD	articles	author	483	9836	7440
NITF-100	body.content	block	3476	3476	4411
NITF-1	body.content	block	1946	2801	5174

Table 3: Summary of $\mathcal{EE}$-**Join** Queries



(a) Real-World Data Sets



(b) Synthetic Data Sets

Figure 8: Total Elapsed Time of Query $\mathtt{E_A/\_*/E_B}$

top-down and bottom-up methods. The actual queries used in the experiments are summarized in Table 4. Shakespeare data set was not used, because the data set contains no attribute.

The total elapsed times are shown in Figure 9. For a data set with a relatively small number of attributes such as SIGMOD data set, the bottom-up method was expected to outperform the top-down method, because traversing up the tree for a small number attributes can be more efficient than traversing down the tree with many branches. In our experiments, for the SIGMOD data set, the bottom-up method was the best, followed by $\mathcal{EE}$-**Join** very closely, then by the top-down method in Figure 9(a).

For synthetic data sets, however, the number of attributes was much larger than the number of elements. For such data sets, the performance of the bottom-up method degenerated substantially, because it had to look up the parent elements for so many attributes. We can see this from Figure 9(b)-(c). Since attributes are not allowed to have child nodes, the scope of traversal from an element to its child attributes is limited to one level of a tree. Thus, the top-down method was fairly efficient for the synthetic data sets. Nonetheless, the performance of $\mathcal{EA}$-**Join** algorithm was still better than the top-down method. The main reason is that $\mathcal{EA}$-**Join** needs to scan the element list and attribute list only once without traversing trees.

4.2.3 Scalability Test

We carried out scalability tests of the proposed algorithms with a large data set generated for the NITF document type definition. For both element-element and element-attribute joins, we observed that the query processing time increased almost linearly, as the size of XML data increased. This result shows the linear scalability of the proposed algorithms, and provides another evidence that the proposed path-join algorithms can improve the performance of query processing for XML path expressions over the conventional methods by up to an order of magnitude.

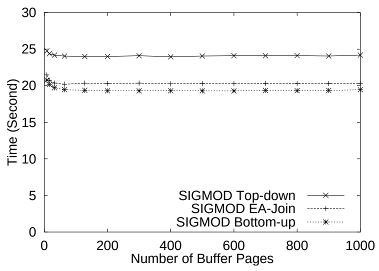
5 Previous Work

For XML databases with graph-based data models, path traversals play a central role in query processing, and optimizing navigational path expressions is an important issue. The optimal query plan depends not only on the *values* in the database but also on the *shape* of the graph containing the data. Three query evaluation strategies have been proposed for Lore's cost-based query optimizer [20]. They are a top-down strategy for exploiting the path expression, a bottom-up strategy for exploiting value predicates, and a hybrid strategy. To speed up query processing in a Lore database, four different types of index structures have been proposed [16, 21]. Value index and text index are used to search objects that have specific values; link index and path index provide fast access to parents of an object and all objects reachable via a given labeled path.

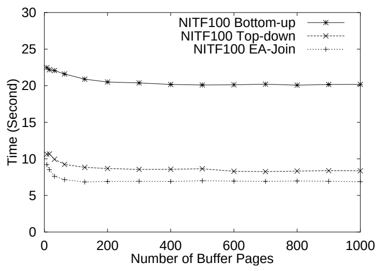
Keyword search is also important to query XML data, if the structures of XML data are not known to users. There have been efforts to integrate keyword search into XML query processing [15, 22]. Florescu and Kossmann [15] propose to extend the XML-QL query language [10] with keyword based search capabilities. To use indexes to facilitate keyword searching, the structure of inverted files is also extended to support full-text indexing with additional information such as the granularity of XML elements, the type of keywords, and the depth of the related element instances. Wolff *et al.* [22] make use of structural information within XML documents in the retrieval process based on a probabilistic model. They propose two index structures: a *structure*

Data Set	Element	Attribute	# of Elements	# of Attributes	# of Results
SIGMOD	author	id	9836	8934	6099
NITF-100	block	dir	3476	25757	2649
NITF-1	block	dir	2801	17152	2127

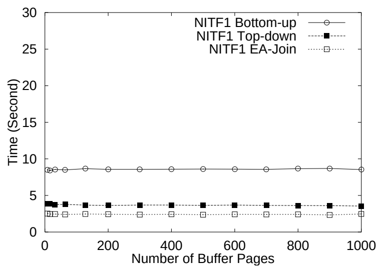
Table 4: Summary of $\mathcal{EA}$-**Join** Queries



(a) SIGMOD data



(b) NITF100 data



(c) NITF1 data

Figure 9: Total Elapsed Time of Query `E[@A]`

index that preserves the hierarchical structure of the underlying data, and a *text index* that supports the evaluation of textual queries.

The problem of optimizing regular path expressions has been studied in the context of navigating semi-structured data in web sites [2, 13]. The semi-structured data is modeled as an edge-labeled graph, where nodes denote HTML pages and edges denote hyperlinks. Abiteboul and Vianu [2] deal with a path query evaluation that takes advantage of local knowledge (*i.e.*, path constraints) about data graphs that may capture structural information about a web site. They address the issue of equivalence decidability of regular path queries under such constraints. Fernandez and Dan Suciu [13] propose two query optimization techniques to rewrite a given regular path expression into another query that reduces the scope of navigation.

New index structures and search algorithms have been proposed for performing efficient filtering of XML documents in the selective information dissemination environments [3]. In such systems, the roles of queries and data are reversed. To effectively target the right information to the right users, user profiles are posed as standing queries that are applied to all incoming XML documents in order to determine which users the document will be sent to. The standing queries are written in XPath language [8], which allows regular path expressions in queries.

To determine the ancestor-descendant relationships, a document tree can be viewed as a complete k-ary tree with many virtual nodes [24]. The identifier of each node is assigned according to the level-order tree traversal. Then, the ancestors and children of a node can be calculated using just the identifier. The problem of this approach is that when the arity and height of the complete tree are getting large, the identifier may be a huge number. For example, for a 10-ary complete tree with a height of 10, the total node number will be around 11 billion, which is too large to store in a four-byte word integer. This makes the approach unrealistic for large XML documents. In [18], the "tree location address" locates a node in a tree by selecting an ancestor node at each level of the tree. So each identifier of an ancestor node is a prefix of its descendants. Using this method will take more space to store identifiers, and the time to determine the ancestor-descendant relationship is not constant. It depends on the length of identifiers.

A recent work has proposed to use the position and depth of a tree node for indexing each occurrence of XML elements [25]. For a non-leaf node, the position is a pair of its beginning and end locations in a depth-first traversal order. The containment properties based on the position and depth are very similar to those of the *extended preorder* independently invented and proposed in this paper.

6 Conclusion and Future Work

We have developed the XML Indexing and Storage System (**XISS**) to store and index XML data and to efficiently process regular path expression queries. The proposed numbering scheme based on extended preorder determines the ancestor-descendant relationship between nodes in the hierarchy of XML data in constant time. The numbering scheme can adapt gracefully to the dynamics of XML data objects by allocating a numbering region with extra space. We plan to investigate the use of document type definition (DTD) to determine the size of a numbering region for an element or an attribute.

The major drawback of the conventional methods based on tree traversals is that they may often require an extensive search of XML data trees. To avoid this drawback, we have proposed an innovative approach to processing a regular path expression query, which decomposes a complex path expression into a collection of basic path subexpressions. Each subexpression can be processed either by directly accessing index structures of the **XISS** system or by applying one of the proposed $\mathcal{EA}$-**Join**, $\mathcal{EE}$-**Join** and $\mathcal{KC}$-**Join** algorithms. For a subexpression having a pair of elements, for example, $\mathcal{EE}$-**Join** algorithm performs its processing by a two-stage sort-merge operation. Experimental results from our prototype implementation of **XISS** show that the proposed algorithms can achieve performance improvement over the conventional methods by up to an order of magnitude.

The new query processing paradigm proposed in this paper poses an interesting issue concerning XML query optimization. A given regular path expression can be decomposed in many different ways. Since each decomposition leads to a different query processing plan, the overall performance may be affected substantially by the way a regular path expression is decomposed. Therefore, it will be an important optimization task to find the best way to decompose an expression. We conjecture that document type definitions and statistics on XML data may be used to estimate the costs and sizes of intermediate results.

In the current prototype implementation of **XISS**, all the index structures are organized as paged files for efficient disk IO. We have observed that trade-off between disk access efficiency and storage utilization. It is worth investigating the way to find the optimal page size or the break-even point between the two criteria.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[2] Serge Abiteboul and Victor Vianu. Regular path queries with constraints. In *the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 122–133, Tucson, AZ, May 1997.

[3] Mehmet Altinel and Michael J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 26th VLDB Conference*, pages 53–64, Cairo, Egypt, September 2000.

[4] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0 second edition W3C recommendation. Technical Report REC-xml-20001006, World Wide Web Consortium, October 2000.

[5] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL: A graphical language for querying and restructuring XML documents. In *Proceedings of the 8th International World Wide Web Conference*, pages 93–109, Toronto, Canada, May 1999.

[6] Don Chamberlin, Daniela Florescu, Jonathan Robie, Jrme Simon, and Mugur Stefanescu. XQuery: A Query Language for XML W3C working draft. Technical Report WD-xquery-20010215, World Wide Web Consortium, February 2001.

[7] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources. In *International Workshop on the Web and Databases (WebDB'2000)*, Dallas, TX, May 2000.

[8] James Clark and Steve DeRose. XML Path Language (XPath) version 1.0 w3c recommendation. Technical Report REC-xpath-19991116, World Wide Web Consortium, November 1999.

[9] Robin Cover. The XML Cover Pages. http://xml.-coverpages.org/xml.html, February 2001.

[10] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. In *Proceedings of the 8th International World Wide Web Conference*, pages 77–91, Toronto, Canada, May 1999.

[11] Angel Luis Diaz and Douglas Lovell. XML Generator. http://www.alphaworks.ibm.com/tech/xmlgenerator, September 1999.

[12] Paul F. Dietz. Maintaining order in a linked list. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 122–127, San Francisco, California, May 1982.

[13] Mary Fernandez and Dan Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the 14th Inter. Conference on Data Engineering*, pages 14–23, Orlando, FL, February 1998.

[14] Daniela Florescu and Donald Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical Report 3680, INRIA, Rocquencourt, France, May 1999.

[15] Daniela Florescu, Donald Kossmann, and Ioana Manolescu. Integrating keyword search into XML query processing. In *Proceedings of the 9th International World Wide Web Conference*, Amsterdam, Netherlands, May 2000.

[16] Roy Goldman and Jennifer Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23rd VLDB Conference*, pages 436–445, Athens, Greece, September 1997.

[17] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st VLDB Conference*, pages 562–573, Zurich, Switzerland, September 1995.

[18] W. Eliot Kimber. HyTime and SGML: Understanding the HyTime HYQ Query Language. Technical Report Version 1.1, IBM Corporation, August 1993.

[19] Olivier Liechti, Mark J. Sifer, and Tadao Ichikawa. Structured graph format: XML metadata for describing web site structure. *Computer Networks and ISDN Systems*, 30:11–21, 1998.

[20] Jason McHugh and Jennifer Widom. Query optimization for XML. In *Proceedings of the 25th VLDB Conference*, pages 315–326, Edinburgh, Scotland, September 1999.

[21] Jason McHugh, Jennifer Widom, Serge Abiteboul, Qingshan Luo, and Anand Rajaraman. Indexing semistructured data. Technical report, Stanford University, Stanford CA, February 1998.

[22] Jens E. Wolff, Holger Florke, and Armin B. Cremers. Searching and browsing collections of structural information. In *IEEE Advances in Digital Libraries (ADL'2000)*, pages 141–150, Bethesda, MD, May 1997.

[23] XMLsoft. The XML C library for Gnome. http://-xmlsoft.org/, January 2001.

[24] Yong Kyu Lee, Seong-Joon Yoo, Kyoungro Yoon. Index structures for structured documents. In *ACM First International Conference on Digital Libraries*, pages 91–99, Bethesda, Maryland, March 1996.

[25] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. In *Proceedings of the 2001 ACM-SIGMOD Conference*, Santa Barbara, CA, May 2001.