XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases

MASATOSHI YOSHIKAWA and TOSHIYUKI AMAGASA
Nara Institute of Science and Technology
TAKEYUKI SHIMURA
IBM Japan, Ltd.
and
SHUNSUKE UEMURA
Nara Institute of Science and Technology

This article describes XRel, a novel approach for storage and retrieval of XML documents using relational databases. In this approach, an XML document is decomposed into nodes on the basis of its tree structure and stored in relational tables according to the node type, with path information from the root to each node. XRel enables us to store XML documents using a fixed relational schema without any information about DTDs and also to utilize indices such as the B^+-tree and the R-tree supported by database management systems. Thus, XRel does not need any extension of relational databases for storing XML documents. For processing XML queries, we present an algorithm for translating a core subset of XPath expressions into SQL queries. Finally, we demonstrate the effectiveness of this approach through several experiments using actual XML documents.

Categories and Subject Descriptors: I.7.1 [**Document and Text Processing**]: Document and Text Editing—*Document management*; I.7.2 [**Document and Text Processing**]: Document Preparation—*Markup languages*; *XML*; H.2.1 [**Database Management**]: Logical Design—*Schema and subschema*; H.3.6 [**Information Storage and Retrieval**]: Library Automation—*Large text archives*

This work was partially supported by the Ministry of Education, Culture, Sports, Science and Technology, Japan, under grants 11480088, 12680417, 12208032, and by the CREST of JST (Japan Science and Technology).
An earlier version of this paper appeared in the *Proceedings of the Tenth International Conference on Database and Expert Systems Applications*, Aug. 30–Sep. 3, 1999 (DEXA'99), pp. 206–217.
Authors' addresses: M. Yoshikawa and T. Amagasa, Graduate School of Information Science, Nara Institute of Science and Technology, 8916-5 Takayama, Ikoma, Nara, 630-0101, Japan; email: yosikawa@is.aist-nara.ac.jp; amagasa@is.aist-nara.ac.jp; T. Shimura, IBM Japan, Ltd., 1623-14, Shimotsuruma, Yamato, Kanagawa, 242-8502, Japan; email: takeyus@jp.ibm.com; S. Uemura, Graduate School of Information Science, Nara Institute of Science and Technology, 8916-5 Takayama, Ikoma, Nara, 630-0101, Japan; email: uemura@is.aist-nara.ac.jp.

General Terms: Algorithms, Design, Management, Performance

Additional Key Words and Phrases: text markup, text tagging, XML query, XPath

1. INTRODUCTION

XML (Extensible Markup Language) [World Wide Web Consortium 1998] is emerging as a standard format for data and documents on the Internet. Various kinds of applications that use the XML format have been developed (see XML Cover Pages <http://xml.coverpages.org>). As a result, many kinds of data will be exchanged in the form of XML documents or XML data. It is expected that not only organizations but also individuals will use large quantities of XML documents in the near future. Developing techniques for storing massive XML documents and retrieving information from them is one of the core problems at the point of contact for research in the database area and XML.

In this article we describe XRel, a novel approach to building XML databases on top of off-the-shelf relational databases. The design goals of XRel are as follows: (1) no restriction should be imposed on the XML documents being stored; any valid or well-formed XML documents should be stored and queried; (2) XML queries should be based on W3C standardization activities; (3) storage and manipulation of XML documents should be made possible using currently available relational databases; no extension of data model, query expressive power, or index structures of relational database systems should be assumed; and (4) query processing should be efficient.

One of the important features of XML documents is that we can perform operations based on their logical structures. Hence, databases that manage XML documents have to support queries on their logical structures and contents. Since access is based on logical structure, it is appropriate to decompose and store XML documents according to their tree structures, which are then stored in databases. In order to retrieve XML documents from such databases, XML queries are translated into database queries (typically in SQL).

There are two approaches to designing database schemas for XML documents, as follows.

Structure-mapping approach: Database schemas represent the logical structure (or DTDs if they are available) of target XML documents. In a basic design method, a relation or class is created for each element type in the XML documents (e.g., Christophides et al. [1994]; Abiteboul et al. [1997]). A more sophisticated mapping method has also been proposed, whereby database schemas are designed based on detailed analysis of DTDs [Shanmugasundaram et al. 1999]. In the structure-mapping approach, a database schema is defined for each XML document structure or DTD.

Model-mapping approach: Database schemas represent constructs of the XML document model. In this approach, a fixed database schema is used to store the structure of all XML documents. Early proposals of this approach include Zhang [1995]. Another example includes the "edge approach" [Florescu and Kossmann 1999b], in which edges in XML document trees are stored as relational tuples.

In this article we adopt the model-mapping approach for the following two reasons:

—The data structure of XML documents has richer expressive power than the relational data model or object-oriented data model; more concretely, neither the relational data model nor the object-oriented data model has constructs to express the order or choice ('|') of elements in the element content models in DTDs. This implies that we cannot find a method of structure mapping that maps data structures of XML documents into database schemas in a natural way. To cope with this problem, we need to extend the expressive power of database models [Christophides et al. 1994 ; Abiteboul et al. 1997]. However, storage schemes assuming extended database models are not applicable to off-the-shelf database systems.

—The structure-mapping approach is suitable when we store a large number of XML documents that conform to a limited number of document structures or DTDs, and when the document structures or DTDs are static. However, numerous sophisticated Web applications are based on the flexible and dynamic usage of XML. In such applications, there is a demand to store various kinds of XML documents: (i) those whose DTDs are not known beforehand, or (ii) those that are well-formed but do not have DTDs. Further, many such applications deal with XML documents whose logical structure changes often. Obviously, the structure-mapping approach is inappropriate for storing a large number of such dynamic and structurally-variant XML documents.

In both of the approaches to database schema design above, XML documents are decomposed into fragments composed of logical units. Obviously, these decomposition approaches have drawbacks—it takes time to restore the entire or a large subportion of the original XML documents, and processing certain text operations such as a proximity search beyond the boundaries of elements becomes very complex. A simple alternative approach to overcome these problems is to store the entire text of XML documents in a single database attribute as a CLOB (Character Large OBject), or as files outside database systems. In our system, we optionally keep the entire text of XML documents as well as their fragments stored in database schemas. The decision whether to keep entire XML documents depends on the demands of the application that will be used. The entire text of each XML document in our system is stored as CLOB data if CLOB is supported in the database system, and is stored as text files otherwise.

There are other important design choices as well: e.g., which database models should we use for the XML document databases—the (object) relational database model or object-oriented database? We chose the (object) relational database for the following two reasons:

—Current use of (object) relational databases is widespread [Leavitt 2000]. Consequently, large quantities of non-XML data have already been stored in them. In order to work in closer cooperation with such traditional data, it is useful to store XML data in the same kind of databases.

—Query optimization techniques and the processing mechanisms in relational databases have been studied for a quarter of a century, and have reached full growth. Thus, it is pragmatic to be able to cope with them.

Many query languages for XML documents had already been proposed [Fernandez and Siméon 1999; Bonifati and Ceri 2000]. Among others, XQL [Robie et al. 1999; Robie 1999] and XML-QL [Deutsch et al. 1998; 1999] are important, in that detailed language specifications are defined for these languages. Quilt [Robie et al. 2000; Chamberlin et al. 2000] is another notable language that integrates the features of many other languages including XQL and XML-QL. Although it was not designed as a full-fledged query language, XPath [World Wide Web Consortium 1999] is a language for addressing parts of an XML document. Standardization of XML query facilities is ongoing at the W3C [World Wide Web Consortium 2001]. In our research, we have adopted XPath as a query language for the following reasons: (i) XPath is a W3C Recommendation and (ii) the functionality of XPath is covered by the expressive power of many query languages proposed thus far, and is mandatory for the future standard query language [World Wide Web Consortium 2000b]. To provide an XML query interface of databases, we need to develop algorithms translating XML queries into database queries (in SQL or OQL).

The data structure of XML documents is modeled by a tree [World Wide Web Consortium 2000a]. If we design a relational schema on the basis of a model-mapping approach, one of the main problems is how to map basic constructs in the tree model to a (object) relational schema. Several approaches have been proposed thus far. For example, one approach is to store edges, as suggested in Florescu and Kossmann [1999b], the other is to store nodes. In Zhang [1995], all text nodes in SGML documents are managed with a `NODE` class. One of the essential differences between our proposal and previous research is that we represent the XML tree structure in terms of a combination of path and region. More precisely, we enumerate available paths from the root to each node in the tree structure of an XML document, and store the path expressions themselves in a relational attribute. We represent the tree structure by combining those pieces of information with information on region. In general, queries on XML documents frequently contain path expressions. Our approach has a clear advantage in processing such queries—that is, we can process them in terms of string matches because every possible path expression is stored in the databases as a string. This feature gives the following quantitative and qualitative advantages:

(1) Our approach has a quantitative advantage in that most of the other approaches, including those of Shanmugasundaram et al. [1999]; Florescu and Kossmann [1999b; 1999a], require join operations in proportion to the length of the paths to process them. If a path expression contains //, the number of join operations becomes as long as the length of the longest path matching the path expression in the document tree. Our approach performs the same processing using string matches. Hence, we can reduce the number of join operations and achieve efficient query processing.

(2) Our approach has a qualitative advantage in that most other approaches, such as those of Shanmugasundaram et al. [1999]; Florescu and Kossmann [1999b; 1999a], require the recursive queries in the database query languages, while XRel does not require recursive queries, and can perform the same function within the SQL-92 standard.

In addition, this article makes the following contributions:

(1) XRel does not require any special indexing structures, and can utilize conventional indexing structures such as the B^+-tree and the R-tree, provided by database systems.

(2) We give an algorithm to transform XPath expressions into SQL queries. To the best of our knowledge, no previous study has provided a complete algorithm for translating query expressions that conform to an XML standard into SQL queries. In addition, some techniques categorized as structure-mapping approaches need to develop algorithms that translate XML queries into database queries for different XML document structures or DTDs. In such cases, query translation itself becomes difficult. Such algorithms are not viable in the approach [Shanmugasundaram et al. 1999]. By contrast, our database schema is simple and uniform, and hence our translation algorithm is also independent of document structures.

(3) Finally, we show the advantages of our approach through some experiments using actual XML documents.

The rest of this article is organized as follows. Section 3 briefly overviews XML documents. Section 4 describes how to store XML documents using relational databases in XRel. Section 5 shows the retrieval of XML documents. Implementation and evaluation of XRel are described in Section 6. Section 7 concludes this paper and discusses future work.

2. RELATED WORK

Because SGML (Standard Generalized Markup Language) [ISO 1986] was a predecessor of XML, there were several studies on the management of structured documents even before XML emerged [Baeza-Yates and Navarro

1996; Sacks-Davis et al. 1994]. Here, we show related work concerning storage and indexing methods for structured documents.

2.1 Storing Structured Documents in Databases

In this section we describe some methods for storing structured documents in databases. These methods can roughly be classified into two categories: a database schema designed for documents with DTD information and storage of documents without any information about DTDs. The latter approaches are capable of storing well-formed XML documents that do not have DTDs. For both approaches, queries on XML documents are converted into database queries before processing.

2.1.1 *Designing Database Schemas Based on DTDs.* First, there are simple methods that basically design relational schemas or object database classes corresponding to every element declaration in a DTD; e.g., see Christophides et al. [1994]; Abiteboul et al. [1997]. Other approaches design relational schemas by analyzing DTDs more precisely.

Shanmugasundaram et al. [1999] proposed an approach to analyze DTD and automatically convert it into relational schemas. In this approach, a DTD is simplified by discarding the information on the order of occurrence among elements. Thus, the simplified DTD preserves only the semantics of child elements concerned with whether the element (a) can occur only once or more times and (b) is mandatory or not. The graph based on the simplified information is called a DTD graph. In order to transform a DTD graph into relational schemas, two techniques, called *Shared* and *Hybrid*, were proposed. In the *Shared* technique, relations are created for all elements in the DTD graph the nodes of which have an in-degree greater than one. Nodes with an in-degree of one are inlined in the parent node's relation. For each element node with an in-degree of zero, a separate relation is created because they are not reachable from any other node. In the DTD graph, edges marked with '*' indicate that the element of a destination node can occur more than once. For each such element, a separate relation is created because relational databases cannot store set values as they are. Finally, element nodes, which appear along with the directed paths from the element in the DTD graph that creates the relational schema (R), are also inlined as an attribute in the relational schema R. However, the directed paths must not contain '*'. In the *Hybrid* technique, elements with in-degrees greater than 2 are also inlined if they are reachable without passing '*'. Incidentally, order information among elements that is discarded in the first step can be represented by adding positional information in the relational schema.

2.1.2 *Storing Structured Documents Without Information about DTD.* There have been several studies that used fixed relational schemas to store structured documents. For example, Horowits and Williamson [1986] proposed storing structured documents (ordered trees) by decomposing them into relational tables. Also, in a study by Zhang [1995], a method to manage

SGML documents using object-oriented database systems was proposed. In that work, all text nodes were maintained by a class NODE. In addition, Florescu and Kossmann [1999b; 1999a] proposed several relational schemas and analyzed their performance. The method proposed in this article differs from previous ones in that information about paths from the root to each node and its position in the document is maintained in relational tables. In addition, our proposal does not impose any prerequisites on the XML documents to be stored, whereas Florescu and Kossmann [1999b; 1999a] assume that each element has an ID attribute.

2.2 Other Approaches

Several studies such as PAT [Salminen and Tompa 1994]; Burkowski [1992]; Clarke et al. [1995a; 1995b] and Navarro and Baeza-Yates [1997] on index files for structured documents have appeared. Sacks-Davis et al. [1998] categorized such indexes into position- and path-based indexes. In position-based indexes, queries are processed using word element and position. On the other hand, paths in tree structures are used in path-based indexes. We do not use special indexes for structured documents in this article; but our storage method is closely related to the concept of the aforementioned indexes.

Finally, the topic of an abstract data type is related to both storage and query retrieval. Blake et al. [1995] describe an approach in which an XML document is regarded as just a sequence of characters, then operations on tree structures are replaced by those on character strings, and an abstract data type is defined in a database having such operations. Our approach differs from previous ones in that we simply use an off-the-shelf database system; that is, we do not need any special full-text search system or indexing structure to translate XPath queries into SQL.

3. AN OVERVIEW OF XML DOCUMENTS

An XML document consists of three parts: an XML declaration, a DTD (Document Type Definition), and an XML instance.[1] An XML declaration and a DTD are not mandatory for an XML document. An XML declaration specifies the version and the encoding of the XML being used. A DTD is a schema that constrains the structure of XML instances and corresponds to an extended context-free grammar. An XML instance is a tagged document. We omit concrete descriptions of an XML declaration and a DTD.

An XML instance is a hierarchy of elements, the boundaries of which are either delimited by start-tags and end-tags, or, for empty elements, by empty-element tags. Character data between start-tags and end-tags is the content of the element. Figure 1 shows an example of an XML instance. A start-tag is the token that encloses an element type with < and >, and an

[1]Although the term "XML instance" does not appear in the XML Recommendation [World Wide Web Consortium 1998], we use this term to represent XML document data excluding an XML declaration and a DTD.

```
<issue>
   <editor>
      <first>Michael</first>
      <family>Franklin</family>
   </editor>
   <articles>
      <article category="research surveys">
         <title>Comparative Analysis of Six XML Schema Languages</title>
         <authors>
            <author>
               <first>Dongwon</first>
               <family>Lee</family>
            </author>
            <author>
               <first>Wesley</first>
               <middle>W.</middle>
               <family>Chu</family>
            </author>
         </authors>
         <summary>As <keyword>XML</keyword> is emerging ... </summary>
      </article>
   </articles>
</issue>
```

Fig. 1. An XML instance.

end-tag is the token that encloses an element type with </ and >. Elements can nest properly within each other, and the nesting represents logical structure. Attribute names and attribute values can be specified within start-tags.

XML documents have two levels of conformance: valid and well-formed. A well-formed XML document follows tagging rules prescribed in XML. An XML document is valid if it is well-formed and if the document complies with the constraints expressed in an associated DTD.

An XML processor examines whether an XML document is well-formed (or valid). The XML processor is a software module used to read XML documents and provide access to their content and structure. It is assumed that an XML processor is doing its work on behalf of another module, called the application [World Wide Web Consortium 1998].

3.1 Data Model for XML Documents

We employ the XPath data model [World Wide Web Consortium 1999] to represent XML documents. We assume that XML documents are guaranteed to be valid or well-formed by XML processors. Here, we briefly introduce the XPath data model. The full specifications of the data model can be found in World Wide Web Consortium [1999].

In the XPath data model, XML documents are modeled as an ordered tree. There are seven types of nodes. For simplicity, in this article we consider only the following four types of nodes. For each type of node, there is a way of determining a *string-value* for a node of that type. Some types of nodes also have an *expanded-name*. *Document order* is defined for all the nodes in the document and corresponds to the order in which the first character of each node occurs in the XML document. *Reverse document order* is simply the reverse of the document order.

—*Root node*: The root node is the root of the tree. The element node for the document element is a child of the root node. The string-value of the root node is the concatenation of the string-value of all text node descendants of the root node in document order.

—*Element nodes*: There is an element node for every element in the document. An element node has an expanded-name, which is the element type name specified in the tag. Element nodes have zero or more children. The type of each child node is Element or Text. The string-value of an element node is the concatenation of the string-values of all text node descendants of the element node in document order.

—*Attribute nodes*: Each element node has an associated set of attribute nodes. Note that the element node is the parent of each of the attribute nodes; however, an attribute node is not a child of the element node. Attribute nodes have an attribute name and an attribute value. Attribute nodes have no child nodes. The expanded-name and string-value of an attribute node is its name and value, respectively. If more than one attribute of an element node exists, the document order among the attributes is not distinguished. This is because there is no order among XML attributes.

—*Text nodes*: Text nodes have character data specified in the XML Recommendation as a string-value. A text node does not have an expanded-name. Text nodes have no child nodes.

The remaining three types of nodes are namespace nodes, processing instruction nodes, and comment nodes. The discussion in this article will be extended to include the remaining three types of nodes.

Figure 2 is a graphic depiction of the data model instance of the XML document in Figure 1. We call such graphs *XML trees*.

4. DATABASE SCHEMAS FOR STORING XML DOCUMENTS

In this section we describe database schemas for storing XML documents. First, the *basic XRel schema*, which is based on SQL-92 data types, is introduced. Then several variations of the basic XRel schema are described. Some of the variations assume functionalities not supported in SQL-92.

4.1 Basic XRel Schema

Since path expressions frequently appear in XML queries, we use paths as a unit of decomposition of XML trees. For each node excluding the root node in XML data model instances, we store the information on the path from the root node to the node. For example, the path from the root to node 3 (and to node 10, respectively) in Figure 2 can be denoted `#/issue#/editor` (and `#/issue#/articles#/article#/@category`, respectively).

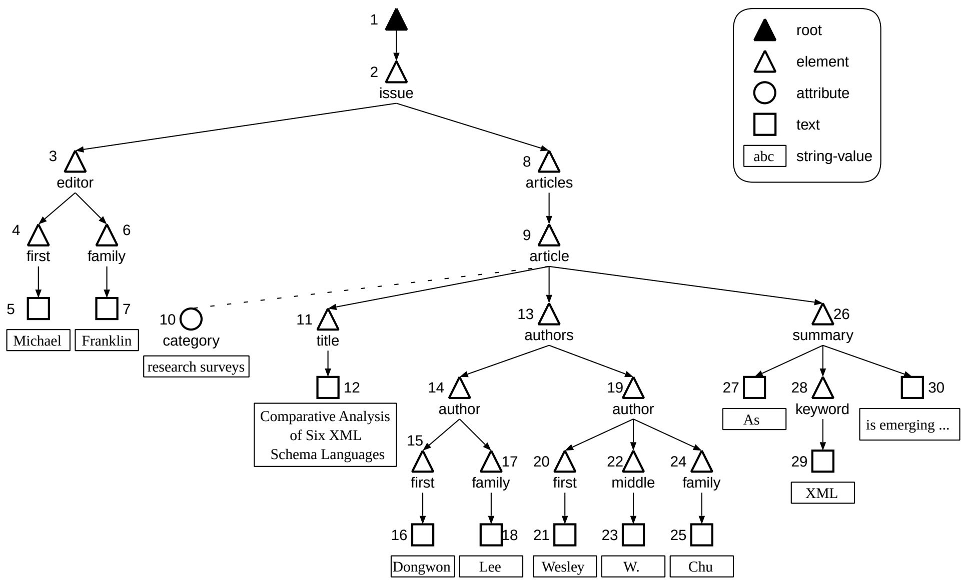

Fig. 2. An XML tree.

More precisely, the path from the root node to an element (or attribute) node can be represented by a path expression defined by the nonterminal symbol `SimplePathExpr` shown in Figure 3. Hereafter, we will call such a path expression a *simple path expression* (of an element node or an attribute node). We also define that the simple path expression of a text node is the same as that of its parent element node. Note that, in simple path expressions, #/ is used as a delimiter of steps, instead of / used in path expressions of XPath. We will explain, in Section 5.1, the reason why we use #/ instead of /.

Unfortunately, simple path expressions are insufficient to restore the topology of XML trees, since more than one node may share the same simple path expression and precedence relationships among nodes are lost in simple path expressions. Hence, to preserve the precedence and ancestor/descendant relationships among nodes, the region of each node is also kept.

Definition 1. The *region* of an element or text node is a pair of numbers that represent, respectively, the start and end positions of the node in an XML document. The *region* of an attribute node is a pair of two identical numbers equal to the start position of the parent element node plus one.

The reason we use a somewhat unusual definition of the region of attribute nodes is twofold: (i) the document order among attribute nodes sharing the same parent element node are left implementation-dependent in the specification of the XPath data model; and (ii) the technical reason that the parent element node of an attribute node can be judged by the comparison $<$ (not by $\leq$) between the first numbers of their regions. The

```
SimplePathExpr ::= '#/' Step
                   | SimplePathExpr '#/' Step

Step           ::= NameTest
                   | '@' NameTest

NameTest       ::= QName
```

Fig. 3. The syntax of simple path expressions.

key idea of the storage scheme in XRel is to keep the combination of the simple path expression and the region of nodes in an XML tree as relation tuples, thus preserving the information of the topology of the XML tree and the expanded-names of nodes.

For example, simple path expressions and the regions of some nodes in Figure 2 can be represented as follows:

node 10	#/issue#/articles#/article#/@category	(82, 82)
node 14	#/issue#/articles#/article#/authors#/author	(190, 248)
node 28	#/issue#/articles#/article#/summary#/keyword	(348, 369).

In the basic XRel schema, we create one relational schema for each node type. A tuple in the relation for a node type represents a node of the type, and stores the simple path expression, the region, and some additional information including the string-value of the node. Nodes of different XML documents are stored in the same relation as long as they are of the same type. To distinguish such nodes, document identifiers are also stored in tuples. Given that there are three node types (element, attribute, and text) excluding the root, we create three relational schemas.

When storing a large number of XML documents having the same or similar structures, which is typically the case when XML documents conforming to the same DTD are stored, a single simple path expression may appear many times in relations. To save the storage space, we replace simple path expressions in the three relations by path identifiers, and create the fourth relation which stores the pair of path identifiers and simple path expressions.

Accordingly, the basic XRel schema consists of the following four relational schemas:

```
Element(docID, pathID, start, end, index, reindex)
Attribute(docID, pathID, start, end, value)
Text(docID, pathID, start, end, value)
Path(pathID, pathexp).
```

The database attributes `docID`, `pathID`, `start`, `end`, and `value` represent document identifier, simple path expression identifier, start position of a region, end position of a region, and string-value, respectively. Given that the occurrence of an element node or a text node is uniquely identifiable by its region, the set of database attributes `docID`, `start`, and `end` is a key of the relation `Element` and `Text`. To identify each of the attribute nodes sharing a common parent element node, an attribute name is required. Given that the suffix of the simple path expression of an attribute node is the attribute name, the set of database attributes `docID`, `start`, `end`, and pathID serve as the key of the relation Attribute. The database attributes index and reindex in the relation Element represent the occurrence order of an element node among the sibling element nodes in document order and reverse document order, respectively. In fact, index and reindex values are not mandatory for restoring original XML documents; however, these values are useful for processing XML queries efficiently. The database attribute pathexp in the relation Path stores simple path expressions.

As an example, Figure 4 shows a database instance of the basic XRel schema that stores the XML document in Figure 1. In Figure 4, *NodeID* right outside of the tables Element, Attribute, and Text is not stored in the tables, but is presented for reference only.

The key features of the basic XRel schema can be summarized as follows: (1) the topology of XML trees and the expanded-name of nodes are represented by the combination of simple path expressions and regions; (2) a relation is created for each node type; and (3) simple path expressions are extracted in a separate relation to reduce the database size.

4.2 Variations of the Relational Schema

The relational schema described above is one of the most basic. Some variations can be considered.

4.2.1 *Schemas Conforming to the SQL-92 Model*.

(1) *Preserving the parent information of each node*: To efficiently process XPath expressions such as /books/*/title, containing a special symbol * that matches any element type, it is useful to keep information about each node's parent, even though information on parents is redundant.

(2) *Alternative methods for representing regions*: In the basic XRel schema, we simply used the number of bytes counted from the beginning of document. Further, we can also consider the following methods:
- —Given an XML document, for a string data we record the word's position in terms of an integer representing its order from the beginning of the document. For a tag, we represent its position in terms of a pair of real numbers; the integral part represents the number of preceding words, and the decimal part represents the number of tags between the previous word and the current tag. Doing this enables us to minimize the effects of the appearance of tags on a word-based proximity search [Sacks-Davis et al. 1998].
- —Generally speaking, the contents of XML documents change as time goes by. When updates to a document occur, positional information about the previous version of that document is no longer useful. Hence, it is important to minimize the effects of document updates on positions. Relative region coordinates (RRCs) [Kha et al. 2001] compute a position in terms of the distance from the beginning of its parent and not from the beginning of the document.

Element

docID	*pathID*	*start*	*end*	*index*	*reindex*	*NodeID*
1	1	0	729	1	1	2
1	2	7	70	1	1	3
1	3	15	36	1	1	4
1	4	37	61	1	1	6
1	5	71	721	1	1	8
1	6	81	710	1	1	9
1	8	118	180	1	1	11
1	9	181	335	1	1	13
1	10	190	248	1	2	14
1	11	198	219	1	1	15
1	12	220	239	1	1	17
1	10	249	325	2	1	19
1	11	257	277	1	1	20
1	13	278	296	1	1	22
1	12	297	316	1	1	24
1	14	336	700	1	1	26
1	15	348	369	1	1	28

Attribute

docID	*pathID*	*start*	*end*	*value*	*NodeID*
1	7	82	82	research surveys	10

Text

docID	*pathID*	*start*	*end*	*value*	*NodeID*
1	3	22	28	Michael	5
1	4	45	52	Franklin	7
1	8	125	172	Comparative Analysis ...	12
1	11	205	211	Dongwon	16
1	12	228	230	Lee	18
1	11	264	269	Wesley	21
1	13	286	287	W.	23
1	12	305	307	Chu	25
1	14	345	347	As	27
1	15	357	359	XML	29
1	14	370	690	is emerging as the ...	30

Path

pathID	*pathexp*
1	#/issue
2	#/issue#/editor
3	#/issue#/editor#/first
4	#/issue#/editor#/family
5	#/issue#/articles
6	#/issue#/articles#/article
7	#/issue#/articles#/article#/@category
8	#/issue#/articles#/article#/title
9	#/issue#/articles#/article#/authors
10	#/issue#/articles#/article#/authors#/author
11	#/issue#/articles#/article#/authors#/author#/first
12	#/issue#/articles#/article#/authors#/author#/family
13	#/issue#/articles#/article#/authors#/author#/middle
14	#/issue#/articles#/article#/summary
15	#/issue#/articles#/article#/summary#/keyword

Fig. 4. A storage example of XML documents.

4.2.2 *Schemas Beyond the SQL-92 Model.*

(1) `TEXT` *type*: If the underlying DBMS supports variable length text data, we can define a relation to store full text data of XML documents in addition to the basic XRel schema. Larger database size is an obvious disadvantage for the augmented database schema; however, executing text search operations becomes possible. Certain proximity searches, such as those across the boundary of elements, will become very fast—although processing these search operations is not feasible in the basic XRel schema.

(2) *Introduction of ADTs (Abstract Data Types)*: If the underlying DBMS supports user-defined ADTs, we can define the database schemas to exploit them. An example of such an ADT is the one representing regions. In the basic XRel schema, two separate attributes are used to represent regions. We can define an ADT `REGION` to manage regions of nodes. An instance of the `REGION` type is a pair of numbers (r, s) such that $0 \leq r \leq s$. Functions that could be useful for this ADT include the following:

—`BOOLEAN contain(REGION pos)`

For a given `REGION` instance `pos` $= (r_a, s_a)$, this function returns `true` if $(r < r_a) \wedge (s_a < s)$ holds, and returns `false` otherwise.

—`BOOLEAN precede(REGION pos)`

For a given `REGION` instance `pos` $= (r_a, s_a)$, this function returns `true` if $s < r_a$ holds, and returns `false` otherwise.

These functions are used to judge the containment and precedence relationships among occurrences of nodes in a document. In addition, if R-tree indices are supported by the DBMS, the processing of these functions is accelerated.

5. QUERY TRANSLATION

In XRel, the relational schema presented in Section 4 is hidden from applications. Users or applications view XML documents modeled as XML trees and issue XML queries against the XML trees. The system then translates XML queries into SQL queries. We describe the query translation algorithm in detail in this section.

Because a standard XML query language has not emerged yet, we focus on a class of query expressions commonly found in the XML query languages proposed so far. An important query construction in XML is path expressions, which often appear in XML queries. XPath is a language for addressing parts of an XML document. Although XPath itself is not a full-fledged query language, its syntax and semantics are used in many proposed query languages. Thus, we focus on a core part of XPath as an XML query language for the XRel. We name the core part *XPathCore*. XPathCore expressions are basically the intersection of the nonterminal symbol `PathExpr` in XPath 1.0 [World Wide Web Consortium 1999] and the

```
Query       ::= PathExpr

PathExpr    ::= RegularExpr
              | '/' RegularExpr
              | '//' RegularExpr
              | BasicExpr Predicate* '/' RegularExpr
              | BasicExpr Predicate* '//' RegularExpr

RegularExpr ::= Step Predicate*
              | RegularExpr '/' Step Predicate*
              | RegularExpr '//' Step Predicate*

Step        ::= NameTest
              | '@' NameTest

Predicate   ::= '[' Comparison ']'

BasicExpr   ::= '(' Comparison ')'
              | Literal
              | Number

Comparison  ::= ArithExpr
              | ArithExpr CompareOp ArithExpr

CompareOp   ::= '='
              | '!='

ArithExpr   ::= BasicExpr Predicate*
              | PathExpr

NameTest    ::= QName
```

Fig. 5. The XPathCore syntax.

nonterminal symbol `PathExpr` in Quilt [Chamberlin et al. 2000]. The syntax of XPathCore is shown more specifically in Figure 5, in a form of simplified syntax of the nonterminal symbol `PathExpr` in Quilt. We assume that queries in XPathCore are given in the form of `PathExpr`. The semantics of XPathCore follow XPath 1.0 [World Wide Web Consortium 1999].

5.1 An Overview of the Query Translation

We give an overview of the translation process from XML queries into SQL queries, against XML documents stored in the relational schema.presented in Section 4. To give an overview of query translation, we begin with the following XPathCore expression:

`/issue//family` (1)

For a given node n in an XML tree, '//' in XPath 1.0 selects the node n and all of its descendant nodes. Hence, query (1) selects all `family` element nodes that are descendants of any `issue` element node. A simple but key observation is that we can easily find the resulting `family` element nodes in XRel databases using string matching in SQL. To give a more general explanation, we define two subclasses of regular expressions (`RegularExpr` in Figure 5) in XPathCore: *simple regular expressions* (`SimpleRegularExpr`

```
SimpleRegularExpr ::= Step
                    | SimpleRegularExpr '/' Step
                    | SimpleRegularExpr '//' Step

SimpleAbsoluteRegularExpr ::= '/' SimpleRegularExpr
                            | '//' SimpleRegularExpr
```

Fig. 6. The syntax of a simple regular expression and a simple absolute regular expression.

in Figure 6) and *simple absolute regular expressions* (`SimpleAbsoluteRegularExpr` in Figure 6). In XRel databases, simple path expressions from the root node to every node are stored (recall the syntax of simple path expression, given in Figure 3). The only difference between simple absolute regular expressions and simple path expressions is that the former have '/' and '//' as a delimiter of steps, while the latter have '#/'. We can find nodes satisfying a simple absolute regular expression *s* by replacing occurrences of '/' in *s* by '#/' and occurrences of '//' by '#%/', and then performing SQL string matching using the string after the replacement against the `pathexp` attribute in relation `Path`. Therefore, the XPathCore expression (1) can be translated into the following SQL query:

```
SELECT   e1.docID, e1.start, e1.end
FROM     Element e1, Path p1
WHERE    p1.pathexp LIKE '#/issue#%/family'
AND      e1.pathID = p1.pathID
ORDER BY e1.docID, e1.start, e1.end
```

We can now explain why we used '#/' instead of '/' as a delimiter of simple path expressions stored in the `Path` relation. If we had stored a path expression in the form `/issue/family`, we would translate query (1) into the above SQL, provided the third line was replaced into the following `WHERE` clause condition.

```
WHERE     p1.pathexp LIKE '/issue%/family'
```

The resulting SQL query returns an incorrect answer because the string pattern

`'/issue%/family'`

matches not only the path expression `/issue/family`, but also other path expressions such as `/issues/family`, `/issuelist/family`, and so on.

Next, let us consider the more complex XPathCore expression below:

`//article[summary/keyword = 'XML']//author/family` (2)

This query retrieves the family names of an article's authors, a summary of which (the article) contains the keyword 'XML'. To translate this query into a SQL query, we need to compensate for the missing prefix of some path expressions in the query. For example, to process the comparison `summary/keyword = 'XML'` in the query using XRel databases, we need to check for the existence of an element node that has a simple path expression satisfying the simple absolute regular expression `//article/summary/keyword`. Likewise, we need to concatenate the path expressions `//article` and `//author/family` to obtain the simple absolute regular expression `//article//author/family`, which, in turn, is translated into the string pattern `#%/article#%/author#/family` in SQL. As this example suggests, in general, simple regular expressions may appear in a query. Also, long simple absolute regular expressions might be divided into fragments by predicates, whereas in the `Path` relation of the XRel databases, simple path expressions are stored. Therefore, to process the general form of path expressions in XPathCore, we need to 'cut' a given query into fragment paths and 'splice' them into the complete form of simple absolute regular expressions. We introduce the *XPathCore graph* to give a clear representation of and guidance for this 'cut and splice' process.

The translation from XPathCore expressions into SQL queries is performed in the following two steps.

(1) In the first step, the XPathCore graph is created as an intermediate representation of XPathCore expressions. An XPathCore expression is divided into simple (absolute) regular expressions. In this process, predicates, groupings, and comparison operators play the role of punctuation marks. Nodes and edges in an XPathCore represent path expressions and their relationships, respectively.

(2) In the second step, SQL queries are generated from XPathCore graphs. SQL clauses are generated for each node and edge in an XPathCore graph.

We give detailed descriptions of each of these two steps in Section 5.2 and Section 5.3, respectively.

5.2 The translation from XPathCore expressions into XPathCore graphs

From the discussion in Section 5.1, we first need to identify the longest possible simple regular expressions and simple absolute regular expressions in a given query. To this end, we begin by presenting an alternative syntax rule for XPathCore expressions. The syntax rule for `RegularExpr` can be rewritten as shown in Figure 7. In this figure, '+' is a meta symbol representing 'one or more occurrences'. From the syntax rule in Figure 7, we can generally represent `RegularExpr` in the following sequence:

$$S_0\{P_1\} \ + \ A_1\{P_2\} \ + \ \ldots\{P_{n-1}\} \ + \ A_{n-1}\{P_n\}*$$

where $n \geq 0$. Also, S_0, A_i $(i = 0, \ldots, n-1)$ and P_j $(j = 1, \ldots, n)$ represent a language of nonterminal symbols: `SimpleRegularExpr`, `SimpleAbsoluteRegularExpr`, and `Predicate` in Figure 5. '{}+' and '{}*' are meta symbols, which represent 'one or many occurrences,' and 'zero or many occurrences,' respectively.

```
RegularExpr ::= SimpleRegularExpr PathStep* Predicate*

PathStep    ::= Predicate+ SimpleAbsoluteRegularExpr
```

Fig. 7. Alternative syntax of regular expressions.

Example 1. For example, query (2) can be viewed as a concatenation of A_0, P_1, and A_1, where A_0 is `//article`, P_1 is `[summary/keyword = 'XML']`, and A_1 is `//author/family`.

To clarify the relationship among simple regular expressions, simple absolute regular expressions, and predicates in a given query, we introduce a graph called the *XPathCore graph*. The formal definition of the XPath-Core graph is as follows:

Definition 2. (*XPathCore graph*). The XPathCore graph is a directed graph $G(N, E)$ satisfying the following constraints:

—Every node has a *node type* that is one of the following seven nonterminal symbols: `BasicExpr`, `Predicate`, `SimpleRegularExpr`, `SimpleAbsolute-RegularExpr`, `Literal`, `Number`, or `Boolean`.

—Every node, other than those of the `Boolean` type, has a *value*. For a node of type T, the value of the node is a language of T.

—N is the union of two mutually-disjoint sets of nodes: *ordinal nodes* and *index nodes*. Ordinal nodes are depicted by a solid circle and index nodes by a dashed circle.

—There is exactly one node in N called the *output node* of G. The output node is depicted by a shaded circle.

—E is the union of two mutually-disjoint sets of edges: E_t (*tree edges*) and E_c (*comparison edges*). Tree edges are depicted by a solid line and comparison edges by a dashed line.

—The graph (N, E_t) is a tree with a root. In (N, E_t), children of a node are ordered. A tree edge from a parent n to its i-th child m is denoted by (n, i, m).

—A comparison edge has a `CompareOp` as a label. A comparison edge from n to m with a label θ is denoted by (n, θ, m).

For example, the XPathCore graph of query (2) is shown in Figure 8. In the figure, the value of a node is depicted near the outside of each node except for n_2, which is a `Boolean` node.

We now explain the major algorithm *GenerateQG*, which produces an XPathCore graph for a given XPathCore expression. From the syntax rule in Figure 5, we can observe that the nonterminal symbols `PathExpr` and `Comparison` are defined in a mutually recursive manner. To simplify the presentation, the algorithm *GenerateQG* is designed for a language of

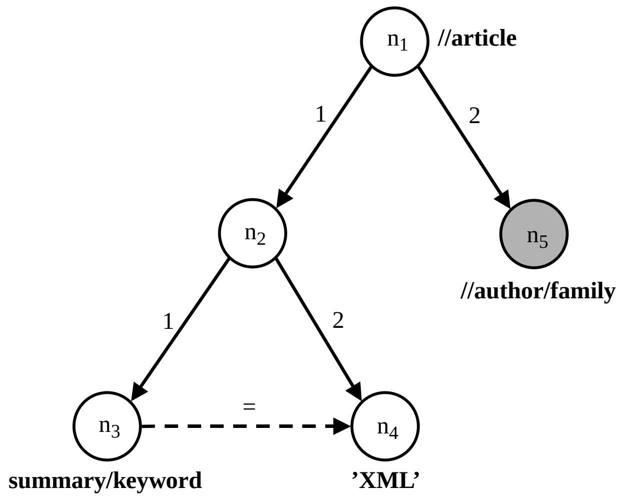

Fig. 8. The XPathCore graph of query (2).

Comparison as an input. Because Comparison has, as its major component, the nonterminal symbol ArithExpr, we first give the algorithm *CreateInitialQG* that returns an initial XPathCore graph for a given language of ArithExpr. Observe that a language of ArithExpr can be represented by one of the following three sequences:

$$B\{P_0\} * A_0\{P_1\} \; + \; A_1\{P_2\} \; + \; \ldots \{P_{n-1}\} \; + \; A_{n-1}\{P_n\}*$$

$$A_0\{P_1\} \; + \; A_1\{P_2\} \; + \; \ldots \{P_{n-1}\} \; + \; A_{n-1}\{P_n\}*$$

$$S_0\{P_1\} \; + \; A_1\{P_2\} \; + \; \ldots \{P_{n-1}\} \; + \; A_{n-1}\{P_n\}*$$

Here, B represents the language of BasicExpr, shown in Figure 5. The algorithm *CreateInitialQG*, given in Figure 9, creates the XPathCore graph in Figure 10. Note that there are two types of P nodes: ordinal nodes and index nodes.

The algorithm *GenerateQG* is given in Figure 11. This algorithm first creates an XPathCore graph in Figure 10 or an XPathCore graph in Figure 12. The algorithm then recursively replaces occurrences of BasicExpr and Predicate with XPathCore graphs, and finally produces an XPathCore graph that contains nodes of five node types:

SimpleRegularExpr, SimpleAbsoluteRegularExpr, Literal, Number, and Boolean.

After creating an XPathCore graph with the algorithm *GenerateQG*, we perform *path concatenation* on the XPathCore graph. In this process we concatenate the value of simple regular expression nodes and simple absolute regular expression nodes along tree edges from the root to a node in an XPathCore graph to obtain a full path expression.

```
CreateInitialQG(E)

Input: ArithExpr $E$
Output: An XPathCore graph $G$

Algorithm:
(1) If $E$ is the following form:
        $B\ \{P_0\} *\ A_0\ \{P_1\}+\ A_1\ \{P_2\}+\ \ldots\ \{P_{n-1}\}+\ A_{n-1}\{P_n\}*$
    then, create an XPathCore graph $G$ depicted in Figure 10. Note that:
    —the root node of $G$ is $B$; and
    —the output node is $A_{n-1}$. (When the sequence $A_0$ and its successor is missing, the output node is $B$.)
(2) If $E$ is the following form:
        $A_0\ \{P_1\}+\ A_1\ \{P_2\}+\ \ldots\ \{P_{n-1}\}+\ A_{n-1}\{P_n\}*$
    then, create an XPathCore graph $G$ depicted in Figure 10, with the following obvious modification:
    (a) the node $B$ and its out-edges are removed; and
    (b) the root node is $A_0$.
(3) If $E$ is the following form:
        $S_0\ \{P_1\}+\ A_1\ \{P_2\}+\ \ldots\ \{P_{n-1}\}+\ A_{n-1}\{P_n\}*$
    then, create an XPathCore graph $G$ depicted in Figure 10, with the following obvious modification:
    (a) the node $B$ and its out-edges are removed;
    (b) the root node $A_0$ is replaced by $S_0$; and
    (c) when the sequence $A_1$ and its successor is missing, the output node is $S_0$.
```

Fig. 9. The algorithm *CreateInitialQG*.

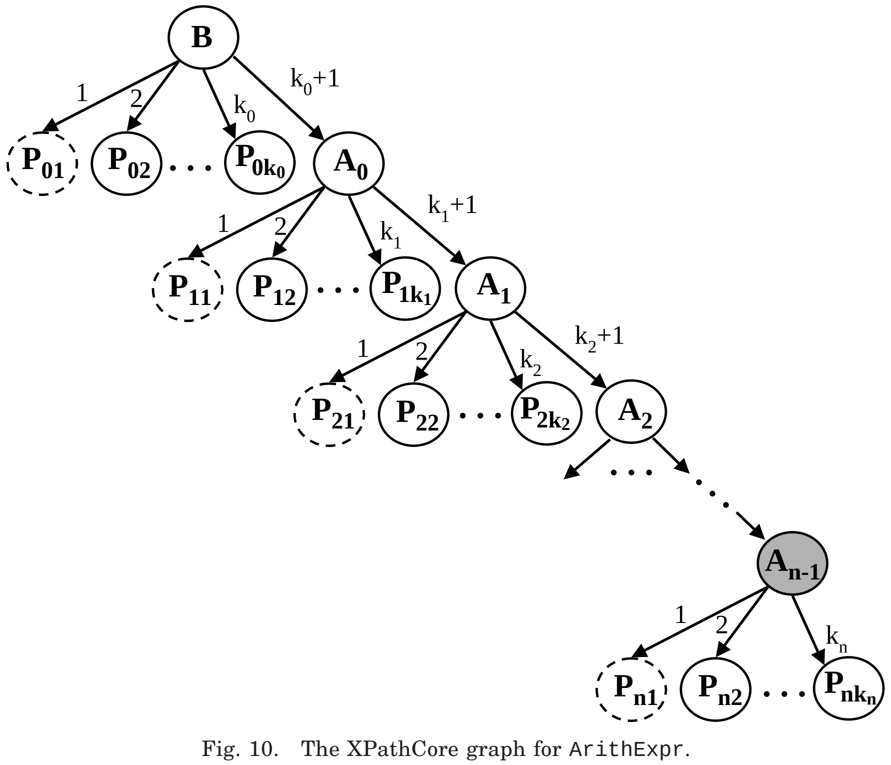

Fig. 10. The XPathCore graph for `ArithExpr`.

```
GenerateQG($C$)

Input: Comparison $C$
Output: An XPathCore graph $G$

Algorithm:
—If $C$ is an ArithExpr $E$:
  (1) $G :=$ CreateInitialQG($E$)
  (2) If there is a node $B$ of type BasicExpr in $G$, perform the followings:
      (a) If the value of $B$ is $(C')$ (where $C'$ is a Comparison):
          Replace the node $B$ in $G$ by a directed graph GenerateQG($C'$). More precisely, do the followings
          in this order:
          i.  $G' :=$ GenerateQG($C'$)
          ii. change $G$ as follows:
              —Let the output node of $G'$ be $O$ and the maximum number of the label of out-edges of
                $O$ be $k_O$ (or 0 if there is no out-edge of $O$.) Change every out-edge $(B, m, X)$ into
                $(O, k_O + m, X)$.
              —Remove $B$.
              —If the output node of the graph before replacement was $B$, let the output node of the graph
                after replacement be $O$. Otherwise, remain the output node unchanged.
      (b) If $B$ is of type Literal: Change the node type of $B$ from BasicExpr to Literal.
      (c) If $B$ is of type Number: Change the node type of $B$ from BasicExpr to Number.
  (3) If there is a $P$ node in $G$, do the followings:
      /* Note that $P$ can not be the output node. */
      Let $P = [C']$ where $C'$ is a Comparison.
      Change the destination of an in-edge of $P$ into GenerateQG($C'$), and remove $P$.
      More precisely, do the followings in this order:
      (a) $G' :=$ GenerateQG($C'$)
      (b) If $P$ was an index node in $G$, let the root node of $G'$ be an index node
      (c) Change $G$ in the following way:
          —Change the destination of an in-edge of $P$ be the root node of $G'$.
          —Remove $P$.
          —Remain the output node of $G$ unchanged (i.e. ignore the output node of $G'$.)
  (4) return $G$
—If $C$ is of the form $E_1$ CompareOp $E_2$ (where $E_1$ and $E_2$ are ArithExpr's):
  Create the following XPathCore graph $G$ and return $G$. (see Figure 12)
  (1) The root of $G$ is a newly created Boolean node $n_b$.
  (2) Let $G_1$ be GenerateQG($E_1$), and $G_2$ be GenerateQG($E_2$). Create the following edges:
      —an edge (with an order label 1) from $n_b$ to the root node of $G_1$; and
      —an edge (with an order label 2) from $n_b$ to the root node of $G_2$.
  (3) Let the output node of $G_1$ be $O_1$, and the output node of $G_2$ be $O_2$. Create a comparison edge from $O_1$
      to $O_2$ with a label CompareOp.
  (4) The output node of $G$ is $n_b$.
```

Fig. 11. The algorithm that generates XPathCore graphs from XPathCore expressions.

Definition 3. Let n be a simple regular expression node or a simple absolute regular expression node in an XPathCore graph G. The *concatenated-value* of n (in G) is defined recursively as follows:

(1) If n has no ancestor simple regular expression or simple absolute regular expression node, the value of n is the concatenated-value of n.

(2) Otherwise, let n_a be the closest ancestor simple regular expression or simple absolute regular expression node. The concatenated-value of n is:
   (a) the concatenation of the concatenated-value of n_a, '/', and the value of n (if n is of simple regular expression type);

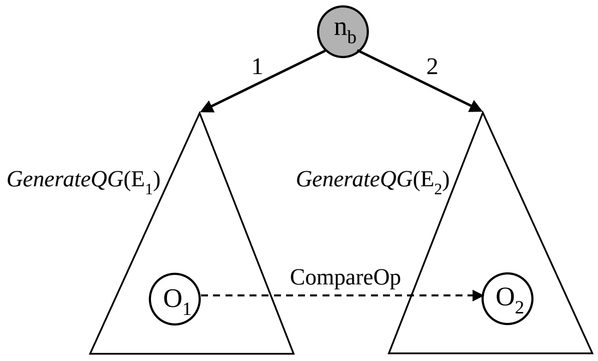

Fig. 12. The XPathCore graph for a comparison of two `ArithExprs`.

(b) the concatenation of the concatenated-value of n_a, and the value of n (if n is of the simple absolute regular expression type).

Example 2. The concatenated-values of the nodes n_1, n_3, and n_5 in Figure 8 are `//article`, `//article/summary/keyword`, and `//article//author/family`, respectively.

5.3 Generating SQL Queries

In this section we present the method to generate SQL queries from an XPathCore graph.

Figure 13 shows the main prcedure of the generation algorithm. The algorithm for generating an SQL query from an XPathCore graph becomes complex when the XPathCore graph contains one or more ordinal nodes of type `Number`. This is because, unlike index nodes, we cannot use the relation attribute `index` to generate SQL queries. Instead, we need to use the `EXIST` and `NOT EXIST` clauses of SQL. As shown in Figure 13, the algorithm first calls the procedure *GenerateSimpleSQL* shown in Figure 15. The procedure *GenerateSimpleSQL* disregards ordinal nodes of type `Number`, and generates an SQL query for the rest of a given XPathCore graph. The main algorithm, *GenerateSQL*, then calls the procedure *ProcessOrdinalNodes* that adds necessary SQL conditions for ordinal nodes of type `Number` in an XPathCore graph.

In Figure 15, we say a node is of type `Expr` if and only if the node is of type `SimpleRegularExpr` or `SimpleAbsoluteRegularExpr`. As illustrated by query 1 in Section 5.1, each occurrence of '/' (and '//', respectively) in path expressions is replaced by '#/' (and '#%/', respectively) and used as a pattern in SQL string matching. To express the replacement formally, we introduce a function $f_{\%}$. For a given `SimpleRegularExpr` or `SimpleAbsoluteRegularExpr` value p, $f_{\%}(p)$ returns a character string obtained by replacing (i) every occurrence of '/' in p by '#/' and (ii) every occurrence of '//' in p by '#%/'.

```
GenerateSQL(G)

Input: An XPathCore graph G
Output: An SQL query

Algorithm:
(1)  Let S be an SQL query GenerateSimpleSQL(G).
(2)  return ProcessOrdinalNodes(S, G).
```

Fig. 13. An algorithm to generate SQL queries.

Example 3. For example, the XPathCore query (2) is translated into the following SQL query.

```
SELECT   e5.docID, e5.start, e5.end
FROM     Path p1, Path p3, Path p5,
          Element e1, Element e5,
          Text t3
WHERE    p1.pathexp LIKE '#%/article'
AND      p3.pathexp LIKE '#%/article#/summary#/keyword'
AND      p5.pathexp LIKE '#%/article#%/author#/family'
AND      e1.pathID = p1.pathID
AND      e5.pathID = p5.pathID
AND      t3.pathID = p3.pathID
AND      e1.start < t3.start
AND      e1.end > t3.end
AND      e1.docID = t3.docID
AND      e1.start < e5.start
AND      e1.end > e5.end
AND      e1.docID = e5.docID
AND      t3.value = 'XML'
ORDER BY e5.docID, e5.start, e5.end
```

Next we turn to two examples of XPathCore expressions, in which the order of elements is specified. The following XPathCore expression retrieves a `family` element that is the second child of an `author` element.

```
//author/family[2]
```

In an XPathCore graph, an index node of type `Number` is created. Then the following SQL query is generated. In this SQL query, a condition on the relational attribute `index` is specified to handle the occurrence order of elements.

```
SELECT   e1.docID, e1.start, e1.end
FROM     Path p1, Element e1
WHERE    p1.pathexp LIKE '#%/author#/family'
AND      e1.pathID = p1.pathID
AND      e1.index = 2
ORDER BY e1.docID, e1.start, e1.end
```

The following XPathCore expression has a similar syntax as, but a different sematics from, the previous expression.

```
(//author/family)[2]
```

This XPathCore expression retrieves a `family` element (i) that is a child of an `author` element and (ii) among elements satifying condition (i) that is second in document order. In an XPathCore graph, an ordinal node of type

```
ProcessOrdinalNodes(S, G)

Input: An SQL query S, and an XPathCore graph G
Output: An SQL query

Algorithm:

(1) Traverse the nodes in the XPathCore graph G in preorder. Let L be the list of ordinal node of type Number
    in G sorted in preorder.
(2) Until L becomes empty do the following:
    (a) Remove the first element m from L.
    (b) Introduce the following notations:
        i              : the value of the node m.
        n              : m's parent node.
        p              : n's parent node, if n has one.
        G_m            : the subgraph of G, consisting of the node n, m's preceding sibling nodes
                         m_prec, and m_prec's all descendants.
        copyF(G_m, k)  : a copy of FROM clauses in S, that is involved solely with nodes in an
                         XPathCore graph G_m. A copy of a variable v in S is named v^k.
        copyW(G_m, k)  : a copy of WHERE clauses in S, that is involved solely with nodes in an
                         XPathCore graph G_m. A copy of a variable v in S is named v^k.
        copyF({p, n^k}) : a copy of FROM clause in S, that is involved with nodes p and n. In this
                         copy, variable e_n^k is used instead of e_n.
        copyW({p, n^k}) : a copy of WHERE clause in S, that is involved with nodes p and n. In this
                         copy, variable e_n^k is used instead of e_n.
    (c) If i ≥ 2, then add the following EXISTS predicate to WHERE clause of the SQL query S.
        AND EXISTS (
           SELECT  *
           FROM    copyF(G_m, 1), ..., copyF(G_m, i − 1),
                   copyF({p, n^1}), ..., copyF({p, n^{i−1}})
           WHERE   copyW(G_m, 1), ..., copyW(G_m, i − 1)
           AND     copyW({p, n^1}), ..., copyW({p, n^{i−1}})
           AND     e_n^1.docID = e_n.docID
           AND     ...
           AND     e_n^{i−1}.docID = e_n.docID
           AND     e_n^1.start < e_n^2.start
           AND     ...
           AND     e_n^{i−2}.start < e_n^{i−1}.start
           AND     e_n^{i−1}.start < e_n.start
        )
    (d) Add the following NOT EXISTS predicate to WHERE clause of the SQL query S.
        AND NOT EXISTS (
           SELECT  *
           FROM    copyF(G_m, 1), ..., copyF(G_m, i),
                   copyF({p, n^1}), ..., copyF({p, n^i})
           WHERE   copyW(G_m, 1), ..., copyW(G_m, i)
           AND     copyW({p, n^1}), ..., copyW({p, n^i})
           AND     e_n^1.docID = e_n.docID
           AND     ...
           AND     e_n^i.docID = e_n.docID
           AND     e_n^1.start < e_n^2.start
           AND     ...
           AND     e_n^{i−1}.start < e_n^i.start
           AND     e_n^i.start < e_n.start
        )
(3) return S.
```

Fig. 14. An algorithm to process ordinal nodes.

```
GenerateSimpleSQL(G)

Input: An XPathCore graph G
Output: An SQL query

Algorithm:
(1) For each node $n$ of type Expr whose concatenated-value is $p$, do the followings:
    —Add "Path p$_n$" in FROM list; and
    —Add "AND p$_n$.pathexp = $f_{\%}(p)$" (if $p$ does not contain '//'); or
      "AND p$_n$.pathexp LIKE $f_{\%}(p)$" (otherwise)
      in WHERE clause.
(2) For each node $n$ which is not connected to a comparison edge, do the followings:
    —If the suffix of $p$, the concatenated-value of $n$, is an element name, then
      —Add "Element e$_n$" in FROM list; and
      —Add "AND e$_n$.pathID = p$_n$.pathID" in WHERE clause.
      Otherwise (i.e. if the suffix of $p$ is an attribute name),
      —Add "Attribute a$_n$" in FROM list; and
      —Add "AND a$_n$.pathID = p$_n$.pathID" in WHERE clause.
(3) For each node $n$ which is connected to a comparison edge, do the followings:
    —If the suffix of $p$, the concatenated-value of $n$, is an element name, then
      —Add "Text t$_n$" in FROM list; and
      —Add "AND t$_n$.pathID = p$_n$.pathID" in WHERE clause.
      Otherwise (i.e. if the suffix of $p$ is an attribute name),
      —Add "Attribute a$_n$" in FROM list; and
      —Add "AND a$_n$.pathID = p$_n$.pathID" in WHERE clause.
(4) For each index node $m$ of type Number in $G$, do the followings:
    —Add "AND e$_n$.index = $k$" in WHERE clause, where $n$ is the parent element node of $m$, and $k$ is the value
      of $m$.
(5) For each pair of two nodes $m$ and $n$ in $G$ such that i) both $m$ and $n$ are a node of type Expr; and ii) $n$ is
    the closest ancestor of $m$, do the followings:
    —Add the followings in WHERE clause:
      —"AND $x_n$.start < $y_m$.start"
      —"AND $x_n$.end > $y_m$.end"
      —"AND $x_n$.docID = $y_m$.docID"
      where
      $x = \begin{cases} \text{e} & \text{(if the suffix of the value of } n \text{ is an element name, and } n \text{ is not connected to a comparison node)} \\ \text{t} & \text{(if the suffix of the value of } n \text{ is an element name, and } n \text{ is connected to a comparison node)} \\ \text{a} & \text{(otherwise, i.e. if the suffix of the value of } n \text{ is an attribute name)} \end{cases}$
      (similar for $y$)
(6) For each comparison edge $(n, \theta, m)$ in $G$,
    —Add "AND $X_n$ $\theta$ $Y_m$" in WHERE clause, where
      $X_n = \begin{cases} \text{t}_n\text{.value} & \text{(if the suffix of the value of } n \text{ is an element name)} \\ \text{a}_n\text{.value} & \text{(if the suffix of the value of } n \text{ is an attribute name)} \\ \text{the value of the node } n & \text{(if } n \text{ is of type Literal or Number)} \end{cases}$
      (similarly for $Y_m$)
(7) For the output node $n$, add:
    —e$_n$.docID, e$_n$.start, e$_n$.end (if the suffix of the value of $n$ is an element name)
    —a$_n$.docID, a$_n$.start, a$_n$.end (if the suffix of the value of $n$ is an attribute name)
    to SELECT clause and ORDER BY clause.
(8) return resultant SQL query.
```

Fig. 15. The algorithm that generates SQL queries from XPathCore graphs.

Number is created. In this case, we need to use EXISTS and NOT EXISTS predicates in the translated SQL query, given below.

```
SELECT   e1.docID, e1.start, e1.end
FROM     Path p1, Element e1
WHERE    p1.pathexp LIKE '#%/author#/family'
AND      e1.pathID = p1.pathID
AND EXISTS (
  SELECT  *
  FROM    Path p11, Element e11
  WHERE   p11.pathexp LIKE '#%/author#/family'
  AND     e11.pathID = p11.pathID
  AND     e11.docID = e1.docID
  AND     e11.start < e1.start
)
AND NOT EXISTS (
  SELECT  *
  FROM    Path p11, Element e11,
            Path p12, Element e12
  WHERE   p11.pathexp LIKE '#%/author#/family'
  AND     p12.pathexp LIKE '#%/author#/family'
  AND     e11.pathID = p11.pathID
  AND     e12.pathID = p12.pathID
  AND     e11.docID = e1.docID
  AND     e12.docID = e1.docID
  AND     e11.start < e12.start
  AND     e12.start < e1.start
)
ORDER BY e1.docID, e1.start, e1.end
```

6. PERFORMANCE EVALUATION

We have implemented XRel and carried out a series of performance experiments in order to check the effectiveness of the method. In this section we report the outlines of the implementation and the experimental results.

6.1 Experimental Setup

We used Sun Enterprise 4000 (4 × UltraSPARC-II 248 MHz CPU, 32GB RAID disk and 2048MB memory) running Solaris 2.5.1 and a commercial relational database system. We utilized IBM's XML4J 3.1.0 (XML parser for Java) on top of Sun JDK 1.2.2 as the XML processor. More specifically, we used the functionalities of the validating XML parser and SAX (Simple API for XML) in order to implement the core module of our system, which converts XML documents into four relations. In that module, we extract every element's simple path expression and its position in the document in a event-driven manner. We then construct the four relations from the results and store them in the database. We used JDBC to connect with the database.

We evaluated the performance of XRel in comparison with other related studies. We selected the study of Florescu and Kossmann [1999a; 1999b] in which XML documents are modeled as ordered and labeled as directed graphs. For simplicity, Florescu and Kossmann do not distinguish elements and XML attributes. Each XML element is represented by a node in the graphs and element-subelement relationships are represented by edges whose labels represent subelement names. Text values are represented as leaves in the graphs.

For their model, Florescu and Kossmann proposed several schemes for mapping XML data into relational tables. They divided the problem into the following two subproblems: how to map edges and how to map values. To solve the former, they proposed the following three approaches:

(E_e) An edge approach that stores all edges of the graph that represents an XML document in a single table.

(E_b) A binary approach that groups all of the same labels into one table.

(E_u) A universal approach that stores a single table containing attributes for all element and attribute names.

To solve the latter, they proposed the following:

(V_s) A separate value tables approach that stores values in separate value tables for each conceivable data type.

(V_i) An inlining approach that stores values and attributes in the same tables.

In theory, we can freely combine the former three approaches for mapping edges and the latter two approaches for mapping values. Florescu et al. carried out a performance analysis on those combinations, and concluded that the combination of (E_b) and (V_i) outperforms the others. This approach is called "binary tables with inlining." In the technique, every XML element is assumed to have a unique identifier, like oid in object databases. All elements with the same name are stored in one table. The table has the following structure:

$$B_{name}(source,\ ordinal,\ value_{int},\ value_{string},\ target).$$

where oids of the source and target elements of each edge are recorded. The key of this table is {*source*, *ordinal*}. Figure 16(a) and (b) show a storage example of Figure 2. In the sequel, we refer to the scheme as FK99. Note that the relational schema in FK99 depends on the structure of XML documents being stored, while XRel does not.

We implemented FK99 using the database on which we implemented XRel. Note that given an XPath query, FK99 requires a number of join operations in proportion to the length of the path expression. Furthermore, a recursive query, which is not supported in SQL–92, is essential when processing '//'. If a recursive query is not supported, we have to expand the query into several subqueries by hand using the information from DTDs.

(a) title

source	ordinal	val_{int}	val_{string}	target
10	2	null	Comparative Analysis of ...	null
...	...	...	...	...

(b) author

source	ordinal	val_{int}	val_{string}	target
13	1	null	null	15
13	2	null	null	17
...	...	...	...	...

Fig. 16. FK99 storage example

6.2 Experimental Results

6.2.1 *Database Size.* We used the Bosak Shakespeare collection[2] as the experimental data. Table I summarizes the characteristics of the collection.

We stored the collection in relational tables using XRel and the approach proposed by Florescu et al., denoted FK99, respectively. Table II (a) and (b) show the sizes of the relational tables. For XRel, four relational tables were generated. The number of tuples contained in each relational table corresponds to the number of nodes in Table I. The *Attribute* table was empty because the test data did not contain any attributes. On the other hand, 22 tables were generated by FK99. In other words, the data contained 22 kinds of elements. More precisely, the largest relation, `SPEECH`, contained 140,277 tuples and the size was 10.2 MBytes. The smallest relation, `SUBTITLE`, contained one tuple and the size was 8 Kbytes. The XRel database size was slightly smaller than that of FK99, but the sizes of both exceed the size of the original document. However, we can say that this is permissible, since the cost of storage devices has greatly declined in recent years and the size of XML documents is usually smaller than that of more complicated data, such as audio and video.

6.2.2 *Query Retrieval.* Tables III and IV show the query set and the time in seconds for processing the queries, respectively. The time was measured for ten runs and the average was recorded. Note that, in many cases, FK99 requires extra processing to reconstruct document fragments from the resulting tuples, since the retrieved answer contains only the identifiers of elements. On the other hand, because XRel keeps the information concerning positions in the original document, all we have to do is extract substrings from the original XML documents.

From the results of Q1 and Q2, we can see that the performance of XRel is not affected by the length of simple path expressions, whereas the performance of FK99 is affected because FK99 requires a number of join operations in proportion to the length of a path expression. On the other

[2]⟨URL: http://metalab.unc.edu/bosak/xml/eg/shaks200.zip⟩

Table I. Test Data Details

#of documents	37
Total size (MB)	7.65
Average size (KB)	206.71
#of element nodes	179,689
#of attribute nodes	0
#of text nodes	147,442
#of simple paths	57

Table II. Database Sizes

(a) XRel		(b) FK99	
Relation	Size (MB)	Description	Size (MB)
Element	10.3	Max	10.2
Attribute	0	Min	0.008
Text	13.2	Average	1.29
Path	0.008	Total	28.29
Total	23.5		

Table III. Performance Evaluation Queries

	Query expression	Feature
Q1	`/PLAY/ACT`	simple path expression (short)
Q2	`/PLAY/ACT/SCENE/SPEECH/LINE/STAGEDIR`	simple path expression (long)
Q3	`//SCENE/TITLE`	one ‘//’
Q4	`//ACT//TITLE`	two ‘//’s
Q5	`/PLAY/ACT[2]`	index
Q6	`(/PLAY/ACT)[2]/TITLE`	grouping and index
Q7	`/PLAY/ACT/SCENE/SPEECH[SPEAKER = 'CURIO']`	text matching
Q8	`/PLAY/ACT/SCENE[//SPEAKER = 'Steward']/TITLE`	‘//’ and text matching

hand, XRel basically processes a path expression in terms of a SQL-92 string match operation. Thus, performance is independent of the length of path expressions.

The queries Q3 and Q4 contain one or two ‘//’. FK99 is faster than XRel for Q3 because a recursive query is not necessary for processing if ‘//’ is at the head of the path expression only. In that case, searching the SCENE table is equivalent to ‘//SCENE’ because the table contains all the SCENE elements in the document. However, if one or more ‘//’ are in the middle of a path expression, FK99 consumes much time, as we can see in Q4. Even so, XRel is effective because XRel can process a ‘//’ as a string match operation, including a wild card (‘%’).

Q5 and Q6 concern predicates using an index operator. Basically, XRel is effective because the information is represented in terms of *index* and *reindex* attributes. However, the information becomes useless if grouping operator ‘()’ is used. In that case, we have to use subqueries with an SQL-92 NOT EXISTS predicate to extract elements in the required order.

Table IV. Query Performance (in seconds)

	XRel	FK99	tuples
Q1	0.021	0.026	185
Q2	0.024	0.694	618
Q3	0.320	0.125	750
Q4	0.304	16.509	766
Q5	0.805	0.159	37
Q6	2.790	0.737	74
Q7	2.748	19.306	4
Q8	9.687	—	6

For this reason, Q6 consumes more time than Q5. In FK99, the same situation holds, but it is faster than XRel.

Q7 and Q8 contain more complicated predicates such as text matching. XRel is still effective, whereas FK99 consumes more time or gives up the processing as translated SQL queries become more complicated. This is mainly due to growing numbers of join operations.

7. CONCLUSIONS

In this article we described XRel, an approach to the storage and retrieval of XML documents that uses (object) relational databases. Using XRel enabled us to easily construct an XPath interface on top of the (object) relational databases.

In this research, we limited extensions to types and functions, and did not need any special indexing structure for query processing. However, some extensions may be needed; for example, abstract data types for synthesizing query results would be required if we were to implement an XML–QL interface. Further, because our approach does not use a special full-text search system, it may not achieve high performance on query retrieval. Thus it is important to develop abstract data types to improve performance. Full-text search for document content, consideration of data types and XML schemas, and support for document updates will be included in our future work.

In some DTDs or document structures, it might be effective to design relational schemas combining the structure-mapping approach and the model-mapping approach. To design optimal relational schemas based on such statistical characteristics of XML documents is a challenging subject of research.

In general, the contents of XML documents vary over time. It is quite useful in many applications to record the temporal changes made to XML documents. In order to capture such temporal XML documents, we are investigating temporal extensions to XML databases [Amagasa et al. 2000; 2001].

REFERENCES

Abiteboul, S., Cluet, S., Christophides, V., Milo, T., Moerkotte, G., and Siméon, J. 1997. Querying documents in object databases. *Int. J. Dig. Lib. 1*, 1, 5–19.

AMAGASA, T., YOSHIKAWA, M., AND UEMURA, S. 2000. A data model for temporal XML documents. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications* (DEXA 2000, Sept.). Springer-Verlag, New York, NY, 334–344.

AMAGASA, T., YOSHIKAWA, M., AND UEMURA, S. 2001. Realizing temporal XML repositories using temporal relational databases. In *Poceedings of the Third International Symposium on Cooperative Database Systems for Advanced Applications* (CODAS' 2001, Apr.), 63–67.

BAEZA-YATES, R. AND NAVARRO, G. 1996. Integrating contents and structure in text retrieval. *SIGMOD Rec. 25*, 1 (Mar.), 67–79.

BLAKE, G., CONSENS, M., DAVIS, I., KILPELAINEN, P., KUIKKA, E., LARSON, P.-A., SNIDER, T., AND TOMPA, F. 1995. Text/relational database management systems: Overview and proposed SQL extentions database prototype. Tech. Rep. 95-25. Centre for the New OED and Text Research, University of Waterloo, Waterloo, Canada.

BONIFATI, A. AND CERI, S. 2000. A comparative analysis of five XML query languages. *SIGMOD Rec. 29*, 1, 68–79.

BURKOWSKI, F. J. 1992. An algebra for hierarchically organized text-dominated databases. *Inf. Process. Manage. 28*, 3, 333–348.

CHAMBERLIN, D., ROBIE, J., AND FLORESCU, D. 2000. Quilt: An XML query language for hetergeneous data sources. In *Proceedings of the International Workshop on Web and Databases* (WebDB '2000). Springer-Verlag, New York, NY, 1–25.

CHRISTOPHIDES, V., ABITEBOUL, S., CLUET, S., AND SCHOLL, M. 1994. From structured documents to novel query facilities. *SIGMOD Rec. 23*, 2 (June), 313–324.

CLARKE, C. L. A., CORMACK, G. V., AND BURKOWSKI, F. J. 1995a. An algebra for structured text search and a framework for its implementation. *Computer J. 38*, 1, 43–56.

CLARKE, C. L. A., CORMACK, G. V., AND BURKOWSKI, F. J. 1995b. Schema-independent retrieval from heterogeneous structured text. In *Proceedings of the 4th Annual Symposium on Document Analysis and Information Retrieval* (Las Vegas, NV). 279–289.

DEUTSCH, A., FERNANDEZ, M., FLORESCU, D., LEVY, A., AND SUCIU, D. 1998. XML-QL: A query language for XML. Submission to the WWW Consortium: http://www.w3.org/TR/NOTE-xml-ql/.

DEUTSCH, A., FERNANDEZ, M., FLORESCU, D., LEVY, A., AND SUCIU, D. 1999. A query language for XML. *Comput. Netw. J. 31*, 16,17 (May), 1155–1169.

FERNANDEZ, M., SIMÉON, J., AND WADLER, P. 1999. XML query languages: Experiences and exemplars. Draft, http://www-db.research.bell-labs.com/user/simeon/xquery.ps.

FLORESCU, D. AND KOSSMANN, D. 1999. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Tech. Rep. 3680. INRIA, Rennes, France. http://rodin.inria.fr/dataFiles/FK99.ps.

FLORESCU, D. AND KOSSMANN, D. 1999. Storing and querying XML data using an RDMBS. *IEEE Data Eng. Tech. Bull. 22*, 3, 27–34.

HOROWITS, E. AND WILLIAMSON, R. C. 1986. Sodos: A software documentation support environment—its definition. *IEEE Trans. Softw. Eng. SE-12*, 8 (Aug.), 849–859.

ISO. 1986. Information processing—Text and office systems—Standard General Markup Language (SGML). ISO-8879.

KHA, D. D., YOSHIKAWA, M., AND UEMURA, S. 2001. An XML indexing structure with relative region coordinate. In *Proceedings of the 17th IEEE International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 313–320.

LEAVITT, N. 2000. Whatever happened to object-oriented databases? *IEEE Computer 33*, 8 (Aug.), 16–19.

NAVARRO, G. AND BAEZA-YATES, R. 1997. Proximal nodes: A model to query document databases by content and structure. *ACM Trans. Inf. Syst. 15*, 4, 400–435.

ROBIE, J. 1999. XML query language (XQL). http://metalab.unc.edu/xql/xql-proposal.xml.

ROBIE, J., CHAMBERLIN, D., AND FLORESCU, D. 2000. Quilt: an XML query language. http://www.almaden.ibm.com/cs/people/chamberlin/quilt_euro.html.

ROBIE, J., LAPP, J., AND SCHACH, D. 1998. XML Query language (XQL). http://www.w3.org/TandS/QL/QL98/pp/xql.html.

SACKS-DAVIS, R., ARNOLD-MOORE, T., AND ZOBEL, J. 1994. Database systems for structured documents. In *Proceedings of the International Symposium on Advanced Database Technologies and Their Integration* (Oct.). 272–283.

SACKS-DAVIS, R., DAO, T., THOM, J. A., AND ZOBEL, J. 1997. Indexing documents for queries on structure, content and attributes. In *Proceedings of the International Symposium on Digital Media Information Base* (DMIB '97).

SALMINEN, A. AND TOMPA, F. W. 1994. PAT expressions: An algebra for text search. *Acta Ling. Hungarica 41*, 1-4, 277–306.

SHANMUGASUNDARAM, J., TUFTE, K., HE, G., ZHANG, C., DEWITT, D. J., AND NAUGHTON, J. F. 1999. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of the 25th International Conference on Very Large Data Bases* (VLDB, Edinburgh, Scotland, Sept. 7-10). Morgan Kaufmann, San Mateo, CA, 302–314.

WORLD WIDE WEB CONSORTIUM. 2001. XML query. http://www.w3.org/XML/Query.

WORLD WIDE WEB CONSORTIUM. 1998. Extensible markup language (XML) 1.0. http://www.w3.org/TR/1998/REC-xml-19980210

WORLD WIDE WEB CONSORTIUM. 1999. XML Path language (XPath) version 1.0. http://www.w3.org/TR/xpath

WORLD WIDE WEB CONSORTIUM. 2000. XML Query data model. http://www.w3.org/TR/2000/WD-query-datamodel-20000511

WORLD WIDE WEB CONSORTIUM. 2000. XML query requirements. http://www.w3.org/TR/2000/WD-xmlquery-req-20000815

ZHANG, J. 1995. Application of OODB and SGML techniques in text database: an electronic dictionary system. *SIGMOD Rec. 24*, 1 (Mar.), 3–8.

Received: January 2001; revised: March 2001; accepted: March 2001