Shredding and Querying XML Data Using an RDBMS

Dennis Plougman Buus
Department of Computer Science
Aalborg University
dbuus@cs.auc.dk

Thomas Pryds Lauritsen
Department of Computer Science
Aalborg University
pryds@cs.auc.dk

Jakob Rutkowski Olesen
Department of Computer Science
Aalborg University
jro@cs.auc.dk

ABSTRACT

This article addresses the problems of querying and retrieving XML data stored in a relational database. We discuss insertion of differently structured XML documents into a database with a constant relational schema by using a simple and general shredding strategy. Closely related to this, and discussed in this article as well, is round-tripping which allows us to return all or a fragment of the data in XML format. We also consider querying the relational data using XML Path Language and XML Query language. Furthermore, we detail an implementation of XPath axis steps as SQL statements, and outline an approach for implementing the central FLWOR construct in the XQuery language.

1. INTRODUCTION

The *Extensible Markup Language* (XML) [5] is a widely used standard for storing and exchanging data, especially through the Internet. It features an intuitive structure, which makes it easily readable for humans, and invites even large, complex XML datasets to be represented in a single file.

Although *native XML databases* [18] are built from scratch for the specific purpose of storing and querying XML documents, they are not yet commonly used. Since much research has been done in the area of *relational databases* [17] compared to native XML ones, they are more widely applied and better performing software is available. Thus, it would be interesting to investigate the possibilities of storing XML data in a relational database using a general strategy and retrieving data from that relational database in XML format. The querying and retrieval of data from the database should be based on XML standards, so that from the outside, it would be opaque that the database is actually relational and not a native XML database. In doing so, we take advantage of the strengths of relational databases, such as indices, while preserving an XML view of the data.

Doing this requires implementation of a process known as *shredding*. This is the process of parsing an XML document and inserting the results into a relational database management system (RDBMS). In our case, we have chosen to do this with simplicity and generality in mind. Also, we wanted to be able to insert several XML documents into the same database without first having to examine the structure of the XML document and most likely needing to adjust the relational database schema accordingly. Reverting the relational data into an XML document is called *round-tripping* (for a definition, see Section 3.2). This is highly dependent on the implementation of the shredder, of course, so a round-tripper can never exist without a coherent shredder.

However, since converting XML data into another representation just to convert it back to its original XML representation is rather pointless, we would like to insert another step in-between. *XML Query Language* (XQuery) [3] is a language designed for querying XML data. It features *XML Path Language* (XPath) [8] which is a mechanism for uniquely describing the path of XML elements in a document, and *FLWOR expressions* which let you form actual queries on XML data. The implementation of the needed subset of these, together with a shredder and a round-tripper, allows us to perform queries on our relational database in a way that one would do on XML data in its pure representation and receive the results as an XML document.

The article is structured in the following manner. We start out by describing preliminary knowledge in Section 2, covering the basics of XML and the parsing of it. Section 3 describes the shredding and round-tripping processes, including the algorithms we utilise. Querying using XQuery, XPath, and FLWOR and the implementation hereof is addressed in Section 4. The article is summed up by an evaluation in Section 5, and finally we give our conclusion is Section 6. Acknowledgements and related work are mentioned in Section 7 and in Section 8

2. PRELIMINARIES

In this section we give an introduction to XML and to how basic documents are created. Furthermore, we present a section on how to parse XML documents.

2.1 The XML Data Model

XML is a language based on tags, in quite the same manner as *Hypertext Markup Language* (HTML) [21]. There are however some differences. XML is used to describe data, and focuses on content, as opposed to HTML, which describes how data should be displayed. In XML there are no predefined tags, so tag names are defined by the author of the document. These things result in a quite simple data model, which offers some basic building blocks, from which more complex models can be built.

The basic parts of XML are elements, attributes and character data. An element will have a start tag and an end tag. Everything in-between the start and end tag, is the content of the element. Element content can be simple con-

tent, mixed content, element content, or empty content. An element with simple content contains only character data, mixed content contains both character data and elements, element content contains only elements, and empty content refers to an empty element, which means that it contains no information.

A sample XML document is shown in Figure 1. The document describes a list of two books. The first book is written by `John Doe`, has the title `The life of John Doe`, and contains an attribute `ref` with the value `23462`.

It is required that elements are properly nested, which means that no element may be ended before all of its contained elements are ended. This hierarchical order of an XML document enables us to consider it as a tree. The top level element, which contains the entire document, is called the root node, and the content of the top level element is considered as the root node's subtree. Furthermore the root node is the ancestor of all its content, and the content is referred to as its descendants. This holds for any node at any level which has a subtree. Nodes which are at the same level in the tree and share the same parent are called siblings. Further, elements can also have attribute data, which must be included in the start tag. These will be represented as children in the tree.

In Figure 1 the root node of the document is `<books>`, and its descendants are `<book>`, `<author>` and `<title>` elements.

```
<?xml version="1.0" encoding="UTF-8" ?>
<books>
  <book ref="23462">
    <author>John Doe</author>
    <title>The Life of John Doe</title>
  </book>
  <book ref="23463" edition="2nd">
    <author>Jane Doe</author>
    <title>Great Cookie Recipes</title>
  </book>
</books>
```

Figure 1: Example of an XML document.

2.2 Parsing XML Data

Reading and working with an XML document requires a parser. We considered two different methods of parsing, namely *Simple API for XML* (SAX) [1] and *Document Object Model* (DOM) [20]. SAX is actually just a lexer. It enables you to work on the different parts of the document as it is being sequentially read by the lexer. DOM is a true parser in the sense that it builds a parse tree, which can then be manipulated. DOM lets you work on a tree structure, but the tree must be kept in memory, which might cause problems when handling large XML documents. We have chosen to work with SAX in this project since we need to work with large documents.

3. SHREDDING AND ROUND-TRIPPING

This section will explain the concepts and our use of both shredding and round-tripping. We will explain how we have mapped XML data to the database, and how we implemented it.

3.1 Shredding

Shredding is the process of parsing an XML document and inserting the result into an RDBMS. For this there are several strategies, of which [9,15] describe a few. Some of these strategies propose a relational schema be constructed from a DTD referred to by the XML document that is to be shredded. Our strategy focuses on having a general relational schema, usable for any XML document. That is, no matter the DTD or XML Schema of a document (or whether the document even *has* a defined schema), the relational schema should be able to contain the contents of the document. Also, we want to be able to store several, differently structured documents in the same relational database.

3.1.1 Representing the XML Data

As previously stated, an XML document consists of nodes nested into each other. Hence, it can be represented as a tree structure where the nodes represent XML start tags and leaf nodes represent character data. The tree representation of Figure 1 is thus shown in Figure 2. This tree ignores attributes; the reason for this is explained later in this section.

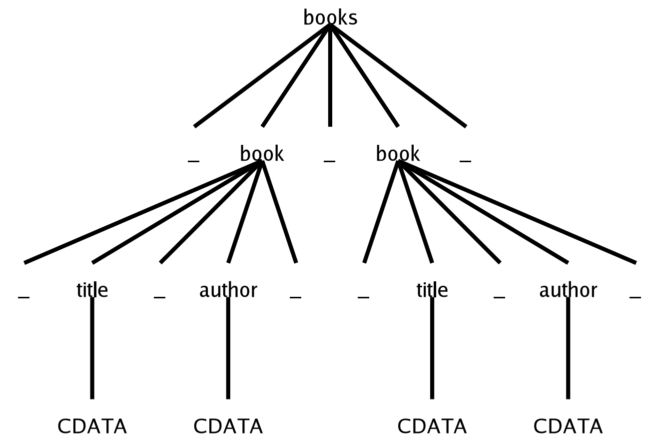

Figure 2: The tree representation of the sample XML document in Figure 1.

Notice that two tags are always separated by character data, which might consist of whitespace only. In this figure, these whitespace-only character data elements are represented by the underscore character (_). For shredding purposes, however, we can safely disregard these. This is illustrated in Figure 3.

3.1.2 Path encoding

The tree representation in Figure 3 can be used to uniquely identify each node and its path by using a Dewey-like classification system [19] where all child nodes of any given node are assigned a (for that set of nodes) unique integer from 1 to n. For each node in the tree, its identification is retrieved by concatenating these integers (separated by slashes) from the root of the tree, following the path to the node in question. As an example, the encircled node in Figure 3 has the path identification "1/2/1/1".

There is a potential problem with this representation, though. In order to be able to store several integers and slashes, an

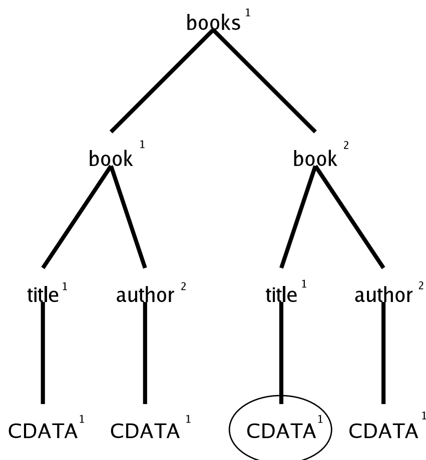

Figure 3: Tree representation disregarding whitespace-only character data.

obvious data type would be a string/varchar. However, if one tries to sort a set of strings, they will be returned in lexical order, whereas for this purpose we really want them returned according to their position in the parse tree. That is, "1/10/1" will be returned *before* "1/9/1", although it ought to come in *after*. Also, there are potential problems concerning the length of the string if one tries to determine the tree depth by measuring the string length instead of making sure to count the number of substrings separated by slashes.

A naive solution to this is to prefix each section between the slashes by zeroes, so that for instance "1/10/1" becomes "000001/000010/000001". This would indeed solve the sorting problem but it introduces a couple of other problems, namely that an XML element can then only contain a certain number (in the case of this example, one million) of children, but more importantly, the zeroes take up a lot of space. Using the string length of the example above, for any set of sibling nodes, all nodes from 1 to 9 yield an overhead of $\frac{5}{6}$, nodes 10 to 99 yield an overhead of $\frac{2}{3}$, etc.

An optimal solution would therefore be required to take up no more space than necessary, it should be indefinitely scalable, and allow sorting. Inspired by the *UTF-8* [2] encoding of Unicode, the encoding of the following user-defined data type satisfies the above requirements. A path step, i.e. an integer, is encoded in chunks of one byte, where the first 7 bits of a byte hold the actual integer or part of it, and the last bit indicates whether the integer continues into the next byte. This is illustrated in Figure 4.

Range	Byte 1	Byte 2	Byte 3
0 to $2^7 - 1$	$xxxxxxx0$		
2^7 to $2^{14} - 1$	$xxxxxxx1$	$xxxxxxx0$	
2^{14} to $2^{21} - 1$	$xxxxxxx1$	$xxxxxxx1$	$xxxxxxx0$

Figure 4: Encoding of a path step (an integer) of a path identification.

Always being able to extend a chunk of bytes by an extra byte allows us to hold infinitely large integers, and thereby infinitely many siblings. Defining a data type that holds infinitely many path steps in the same manner would then allow us to hold infinitely deep trees. A path step of value $2^7 - 1 = 127$ takes up three bytes using the naive solution but only one byte using the optimal one. Likewise the value $2^{14} - 1 = 16383$ takes up five and two bytes, respectively. In order to allow sorting, one would have to define a custom comparison function for the data type.

3.1.3 Relational schema

As mentioned, our relational schema is very general, and thus it consists of only two tables; one for XML nodes, and one for XML tag attributes. The first table (Figure 5) has three rows; one for the path ID, a char-column specifying the type (tag element or character data) of a given node, and a column holding the contents of a node. For element nodes, the content is the name of the tag, and for character data nodes, it is the data itself. We use the `pathID` column as primary key for the table, since this is unique for every node.

Nodes		
pathID	**type**	**contents**
1	tag	books
1/1	tag	book
1/1/1	tag	author
1/1/1/1	cdata	John Doe
1/1/2	tag	title
1/1/2/1	cdata	The Life of John Doe
1/2	tag	book
1/2/1	tag	author
1/2/1/1	cdata	Jane Doe
1/2/2	tag	title
1/2/2/1	cdata	Great Cookie Recipes

Figure 5: Relational schema for nodes and example data from shredding the document in Figure 1.

The attribute table (Figure 6) holds attributes from element nodes, if such exist. According to the XML Recommendation [6], the attributes of a tag need not appear in any particular order, so we can disregard them when constructing the tree and assigning node path IDs. In other words, we can think of attributes as an association list where all we need to store, besides the attribute itself, is its association to a given node. Therefore, a column containing the path ID of the owner-node and a column holding the name of the attribute form a joint primary key. `pathID` is a foreign key to the `pathID` column in the nodes table. In addition, we need to store the contents of the attribute in a column, of course.

Attributes		
pathID	**attrName**	**contents**
1/1	ref	23462
1/2	ref	23463
1/2	edition	2nd

Figure 6: Relational schema for attributes and example data from shredding the document in Figure 1.

3.1.4 Shredding Algorithm

A SAX parser [10] enables us to specify what should happen when certain events occur during the parsing of an XML document. In our case, these events are the following: Parsing is initiated, a start-tag is met, an end-tag is met, and character data (excluding character data in attributes) is

met. The algorithm in Listing 1 uses a stack of integers which represents the path ID of the currently reached node.

```
1  // INITIAL ACTIONS:
2  Push document ID to stack
3
4  // START TAG:
5  Assign current content of stack as path ID for
        tag node
6  Save attributes, if any, using content of stack
        as reference
7  Push 1 to stack
8
9  // END TAG:
10 Pop an element from stack
11 Increase top element of stack by one
12
13 // CHARACTER DATA:
14 Assign current content of stack as path ID for
        character data node
15 Increase top element of stack by one
```

Listing 1: Algorithm for shredding.

To reach our goal of being able to store several documents in the same relational database, we initially push a document ID to the stack, uniquely identifying the document (consecutive numbers will do). This will render the first integer of all path IDs of that document uniquely identifyable. Notice also that nothing is stored in the database upon meeting an end tag. This does not mean, however, that we lose information. The positions of an XML document's end tags are implicitly stored in the tree representation and thereby in the path identification so that we can correctly revert the relational data into an XML document.

3.2 Round-tripping

In Computer Science generally, the term "round-tripping" refers to the concept of converting one representation into another one and then back again, [22]. In this article, however, we use it only to refer to the process of converting shredded relational data back into XML documents (i.e. we do not count the shredding process as a part of round-tripping).

3.2.1 Round-tripper Algorithm

In order for the round-tripper algorithm presented in this section to correctly regenerate an XML document (or a fragment of it), it must receive tuples from the relational database sequentially, ordered according to path ID. Listing the tuples in this order is equivalent to traversing the XML tree in-depth, and B-tree [12] indices guarantee this sort order. We can utilise the fact that this is exactly the order in which the tags should appear in the resulting XML document, *and* we can use the path ID to extract end tags in the XML document.

As Listing 2 illustrates, the key to inserting end tags is a stack that holds the nodes from the root of the tree to the current node. By also remembering the previously seen path ID in another variable, we can compare the tree depth of the previously seen node to the current one and thereby decide whether we need to prefix the start tag, that we are going to set, with one or more end tags. When leaving a leaf node, no matter how many levels you ascend, you will only descend by one level (see Figure 7). Therefore, if the previously seen node was a leaf node of character data, the number of needed end tags is the difference between the two tree depths. If the previous node was an XML tag, then it must be closed as well, and you will have to add another end tag from the stack.

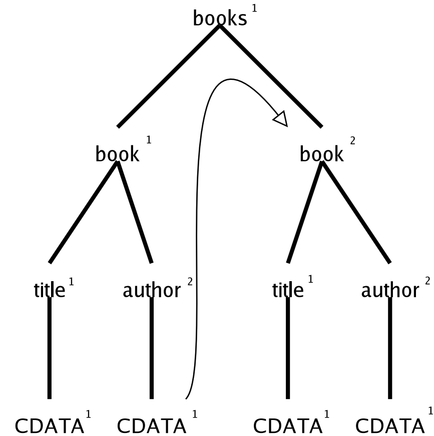

Figure 7: Traversing a tree.

An iteration of the algorithm ends by writing the actual current node to the document as either a start tag or character data. Note, that at this point the algorithm must know the potential attributes of a start tag already. As previously stated, the order of these is not significant, though.

3.2.2 Canonical XML Equivalence

After (correctly) shredding and round-tripping, you end up with two XML documents (input and output) that are logically equivalent but may differ in byte-wise comparison. Since attributes are not required to appear in any particular order, the ordering of these may differ in the two documents. Also character representations may differ; the documents may be expressed in different character sets and throughout the documents different kinds of character escaping may be used.

Canonical XML [4] is a syntax for unambiguously expressing an XML document, and it may be used to show the correctness of a shredded and round-tripped document since it must represent the same logical structure as its source document. After canonicalizating both the input and output documents, their canonical forms may be compared byte-by-byte, and if they prove identical the original documents are said to be equivalent (see Figure 8).

4. QUERY PROCESSING

This section deals with the topic of processing queries over XML documents. Two issues will be examined, namely XML Path Language [8] and XML Query Language [3]. XML Path Language (XPath) is a language in it self, but also an integrated part of XML Query Language (XQuery). We start this section by introducing XPath, examining a number of issues regarding it's implementation, and then go on to do the same for XQuery.

```
For all node tuples in alphanumerical order, ordered by Path ID
        If current node is closer to the root or at the same level as the previously seen node
                Repeat (previous node path depth - current node path depth) times
                        Pop node from stack and write it as XML end tag
                If previously seen node was of type tag
                        Pop an extra node from stack and write it as XML end tag
        If type of current node is tag
                Write current node as XML start tag including its attributes, if any
                Push current node unto stack
        Else if type of current node is character data
                Write out data as text
While stack is not empty
        Pop node from stack and write it as XML end tag
```

Listing 2: Algorithm for round-tripping.

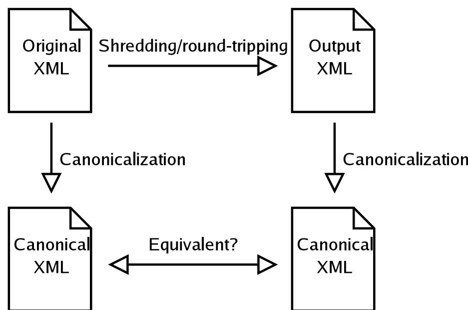

Figure 8: Canonical XML equivalence.

ancestor::	ancestor-or-self::
attribute::	child::
descendant::	descendant-or-self::
following::	following-sibling::
namespace::	parent::
preceding::	preceding-sibling::
self::	

Figure 9: XPath axes.

4.1 XML Path Language

As mentioned, XML documents may be represented as trees of nodes. XML Path Language is used to select specific nodes in such a tree. An example XPath expression which selects the names of all authors who have written a book, from the XML document in Figure 1, might look like this:

`//book/author`

An XPath expression is resolved in a step-wise manner, from left to right. The / element is a step divider, and an expression starting with a / refers to an absolute path, beginning from the root node. The presence of // in an expression refers to all descendant nodes in a document from a given point. It is also possible to refer to specific attributes by using `@`, for example `//@ref`, which selects all ref attributes in the document, and boolean predicates, written like `[author="John Doe"]`, may also be used in an XPath expression to further narrow the results.

Central to XPath is the notion of axes. An axis defines a relation between a single node and a number of related nodes. The available axes are shown in Figure 9. In the example expression above, we used the abbreviated notation, which does not contain explicit axis notation. The unabbreviated version of our example, however, would be:

`/descendant-or-self::book/child::author`

A construct such as `child::author` is an example of a single axis step. An axis step consists of an axis name and a node test. The axis name denotes the relationship between the resulting set of nodes and a context node, and the node test filters the results. Further on in this article we use two special node tests; `text()` and `node()`. `text()` simply selects all text nodes, and `node()` selects any type of node that an XML document may have. Every axis step must, obviously, include an axis name. In the abbreviated notation, the axis `child::` is implied whenever an axis name is not specified. `//book` is the abbreviated notation for `descendant-or-self::book`.

Strictly speaking, the correct abbreviation of `//book` is:

`/descendant-or-self::node()/child::book`

However, for this example the unabbreviated XPath expression described above is sufficient.

In the example above, the individual steps are:

`descendant-or-self::book` - This selects every element in the document which is a `book`. In this first axis step of the expression, the context node is the root node, and therefore `descendant-or-self::` evaluates to every element in the document. The node test filters out every node which is not a `book`.

`child::author` - This step selects every node which is an `author` and a child of one of the nodes in the result set of the previous step.

The the elements resulting from this example query would be:

```
<author>John Doe</author>
<author>Jane Doe</author>
```

Conceptually, each XPath axis step is evaluated on the set of results from the previous step. More accurately, the results of one step each become the context node for the next step, which is run once for each context node. For the first step, the context node is always the root node. This step-wise evaluation lends itself well to an implementation where we construct the different possible step operators as individual components which may be run in a chain to perform queries.

4.1.1 Implementing XPath

The fact that our path IDs are Dewey encoded, means that they carry with them information about the entire ancestry of the node. Having this information within each node provides us with excellent opportunities for locating nodes along the axes. Therefore, the axis steps may be reduced to pattern matching on the string representation of the Dewey encoded path IDs. What follows is a list of the axes, with a description of their implementation and the actual SQL statements that may be used as building blocks when implementing an XPath evaluator. These procedures include a *c_node*, which is the path ID of the context node, and a *nodeTest*, which is used to limit results to those nodes which match the node test. These SQL statements may be used as the bodies in a set of functions stored in the database for convenient access. The statements assume the table layout described in section 3.1.3.

- `self::` - we return the context node.

  ```
  SELECT * FROM nodes
  WHERE pathID = c_node
  AND contents = nodeTest
  ```

- `parent::` - we remove the last element from the path ID of the context node and select the node which matches the resulting path ID. Removing the last element of a context node path ID such as "000001/000003" is equivalent to moving up one level in the syntax tree, thus selecting the parent of the context node.

  ```
  SELECT * FROM nodes
  WHERE pathID = parent_name(c_node)
  AND contents = nodeTest
  ```

  Note: `parent_name()` is a stored function which removes the last 7 characters from a path ID stored using the naive solution for Dewey encoding described in Section 3.1.2. It is coded thus:

  ```
  parent_name(c_node){ RETURN LEFT(c_node, LENGTH(c_node)
  - 7) }
  ```

- `ancestor-or-self::` - in this procedure we make use of a function `is_prefix(substr, str)` to select all nodes whose path ID are a prefix of the path ID of the context node.

  ```
  SELECT * FROM nodes
  WHERE is_prefix(pathid, c_node)
  AND contents = nodeTest
  ```

  The body of the `is_prefix(substr, str)` function is:

  ```
  is_prefix(substr, str){
    IF LOCATE(substr, str) = 1
      RETURN TRUE
    ELSE
      RETURN FALSE
    END IF
  }
  ```

  `LOCATE()` is a built in MySQL function which returns the position of the first occurrence of a substring within a string. If the path ID of a node being considered is a substring at position 1 of the path ID of the context node, then the node is an `ancestor-or-self::`. Most, if not all, SQL implementations have a similar built in function.

- `ancestor::` - we evaluate this axis in much the same way as `ancestor_or_self()`, except we use the parent of the context node in the argument for `is_prefix()`

  ```
  SELECT * FROM nodes
  WHERE is_prefix(pathid, parent_name(c_node))
  AND contents = nodeTest
  ```

- `attribute::` - we select all entries from the attribute table where the path ID matches that of the context node.

  ```
  SELECT * FROM attributes
  WHERE pathid = c_node
  AND contents = nodeTest
  ```

- `child::` - we select all nodes whose path IDs are a concatenation of the path ID of a context node and one additional path element(*one forward slash and exactly 6 characters*). The underscore matches exactly one character when used with the `LIKE` operator.

  ```
  SELECT * FROM nodes
  WHERE pathid LIKE CONCAT(c_node, '/______')
  AND contents = nodeTest
  ```

- `descendant::` - we select all nodes whose path IDs are a concatenation of the path ID of the context node and an arbitrary number of additional path elements. The percent is a wildcard matching one or more characters when used with the `LIKE` operator.

  ```
  SELECT * FROM nodes
  WHERE pathid LIKE CONCAT(c_node, '/%')
  AND contents = nodeTest
  ```

- `descendant-or-self::` - same as above, adding the context node to the result.

  ```
  SELECT * FROM nodes
  WHERE pathid (LIKE CONCAT(c_node, '/%')
  OR pathid = c_node)
  AND contents = nodeTest
  ```

- `following::` - we select all nodes whose path IDs are alphanumerically greater than that of the context node.

  ```
  SELECT * FROM nodes
  WHERE pathid > c_node
  AND contents = nodeTest
  ```

- `following-sibling::` - as above, limiting the results to nodes whose parent ids are the same as the parent of the context node.

```
SELECT * FROM nodes
WHERE pathid > c_node
AND parent_name(c_node) = parent_name(pathid)
AND contents = nodeTest
```

- `preceding::` - we select all nodes whose path IDs are alphanumerically smaller than that of the context node.

```
SELECT * FROM nodes
WHERE pathid < c_node
AND contents = nodeTest
```

- `preceding-sibling::` - as above, limiting the results to nodes whose parent ids are the same as the parent of the context node.

```
SELECT * FROM nodes
WHERE pathid < c_node
AND parent_name(c_node) = parent_name(pathid)
AND contents = nodeTest
```

- `namespace::` - our implementation omits this axis.

4.2 XML Query Language

XML Query Language (XQuery) is a language similar to SQL. It is used to query XML documents, and shares the same data model as XPath. In this section, we first provide an introduction to FLWOR expressions (pronounced "flower"), which are the fundamental building blocks of many interesting XQuery expressions, after which we detail our implementation.

4.2.1 FLWOR Expressions

XQuery contains a query construct known as FLWOR expressions. Its structure is akin to the `SELECT-FROM-WHERE` construct in SQL, and its name is formed from the first letters in the key words of the construct, namely `for`, `let`, `where`, `order by`, and `return`.

An example of a FLWOR expression which returns the title of every book written by "`John Doe`" from our example XML document in Figure 1 is shown in Figure 10.

```
for $book in //book
let $title := $book/title
where $book[author="John Doe"]
order by $title
return
<book>
 { $title }
</book>
```

Figure 10: An example of a FLWOR expression.

In this example, the `for` clause binds the result of the expression "`//book`" to the variable `$book`, and for each book the `let` clause binds the title to the `$title` variable. The `where` clause filters out any node where the author is not "`John Doe`", and the `order by` clause sorts the results alphabetically by title. Finally, the `return` clause returns a fragment of XML with the title enclosed by "`<book></book>`" tags. The result of running this query on our example XML document (Figure 1) is:

```
<result>
 <book>
  <title>The Life of John Doe</title>
 </book>
</result>
```

On the conceptual level, FLWOR expressions follow the data flow model outlined in Figure 11. Their evaluation may be described as a process of step-wise refinement.

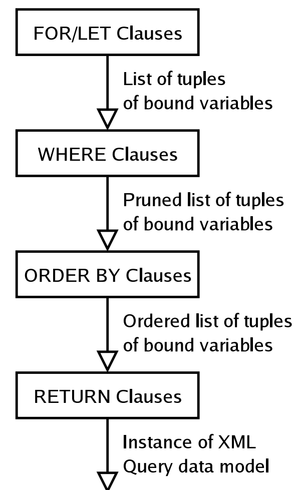

Figure 11: Stages of FLWOR evaluation

The stages of this data flow model may be outlined thus:

- The `for` clauses bind the results of expressions to variables, creating a stream of tuples. Each tuple in the stream contains the variable binding of one of the items in the result of the expression with which it is associated. A `let` clause adds the entire result of its evaluation to each of the tuples created by `for` clauses, if such exist, otherwise it will create a single tuple.
- The stream of tuples is subjected to the `where` clause. This clause filters the tuple stream according to a conditional statement. Only tuples for which the statement holds true will survive. This pruned list of tuples then serves as the input for the next step in the evaluation:
- The `order by` clause applies an ordering to the filtered tuples.
- The `return` clause is responsible for returning the result of the FLWOR expression as XML. For each tuple in the stream, the `return` clause constructs the appropriate fragment of XML, based on the bindings in the tuple. Since `return` must output valid XML, results are packaged inside `<result></result>` tags to ensure that they have a root node.

In the rest of this section, we examine some interesting details regarding the `for/let` stage of FLWOR evaluation.

As we have seen, the `for` and `let` clauses in a FLWOR expression both bind the result of an expression to a variable name, albeit in slightly different ways. The `for` clause binds each element in a result to the variable, iteratively, whereas the `let` clause binds the entire result to the variable. To illustrate the differences between `for` and `let` we present two example queries:

```
for $i in (1,2,3)
return <tuple>{$i}</tuple>
```

The query binds the result of the expression `(1,2,3)` to the variable `$i`, iteratively. This creates a number of tuples, each of which contains the binding of a single item in the result to the variable. The resulting tuples are illustrated in the output of this query:

```
<result>
 <tuple>1</tuple>
 <tuple>2</tuple>
 <tuple>3</tuple>
</result>
```

If we write a similar query, this time using `let`, the result is quite different.

```
let $i := (1,2,3)
return <tuple>{$i}</tuple>
```

The `let` clause binds the entire result of the expression to the variable `$i`, without iteration. Therefore, the query yields just a single tuple:

```
<result>
 <tuple>1 2 3</tuple>
</result>
```

In cases where there are more than one `for` clause in the FLWOR expression, the resulting tuples are the Cartesian product of each variable assigned in a `for`. Consider the query:

```
for $i in (1,2)
for $j in (3,4)
return <tuple>{$i},{$j}</tuple>
```

As before, each variable is iteratively bound to the evaluation of its associated expression. For each iteration of the `$i` variable, a tuple is created for each iteration of the `$j` variable. The resulting tuples are therefore:

```
<result>
 <tuple>1,3</tuple>
 <tuple>1,4</tuple>
 <tuple>2,3</tuple>
 <tuple>2,4</tuple>
</result>
```

As mentioned above, when a `let` clause is included along with a `for` clause, the binding of its variable is added to every tuple. The presence of the `let` clause does not add to the number of tuples.

```
for $i in (1,2)
let $j := (3,4)
return <tuple>{$i},{$j}</tuple>
```

This query creates one tuple for each binding of the `for`-bound variable `$i`. It then adds the binding of the `let`-bound variable `$j` to each of these tuples. The tuple stream created by this query is then:

```
<result>
 <tuple>1,3 4</tuple>
 <tuple>2,3 4</tuple>
</result>
```

4.2.2 Implementing FLWOR

In this section we explore some interesting issues regarding the implementation of FLWOR expressions on our shredded XML documents. We examine issues particular to each of the stages of the conceptual data flow model in the previous section. Moving from the conceptual model to an implementation that may be run on our shredded data within an RDBMS requires some adaptation.

The `for` and `let` clauses

Central to the concept of FLWOR expressions is the aforementioned tuple stream, containing the variables bound in `for` and `let` clauses. This stage of FLWOR evaluation presents the greatest challenges. First, the tuple stream itself will be represented by a table containing one column for each variable. Second, the concept of iteratively bound variables needs to be adapted to the data model of relational databases. The solution is to represent each variable as a table, containing each item in the evaluation of the expression associated with the variable. Thus, a variable `$i` bound in a `for` clause such as this:

```
for $i in (1,2,3)
```

...may be represented by the following table:

$i
1
2
3

The next challenge is in regards to scope. For each `for` and `let` clause, the variables bound in any previous `for` or `let` must be available for use. To illustrate this issue, consider this example XML document:

```
<books>
 <book id="1">
  <author>AuthorA</author>
 </book>
 <book id="2">
  <author>AuthorB</author>
  <author>AuthorC</author>
  <author>AuthorD</author>
 </book>
 <book id="3">
 </book>
</books>
```

Suppose one enters a query starting with these `for` clauses:

```
for $b in //book
for $a in $b/author
```

The point of this query is to create a set of tuples, each tuple containing the binding of an instance of a book and one of its authors. The evaluation of the expression "`$b/author`", associated with the `$a` variable, is clearly dependant on the evaluation of the expression "`//book`", associated with the `$b` variable. Therefore, our goal is to create our tuple stream as a table containing the following tuples:

$b	$a
Book1	*AuthorA*
Book2	*AuthorB*
Book2	*AuthorC*
Book2	*AuthorD*

Note that *Book1* refers to the `<book id="1">` element, and *AuthorA* refers to the `<author>` element containing the text "`AuthorA`". In the actual implementation, the fields in this table will contain the equivalent path IDs instead. Conceptually, the selection of an element contains the entire contents, including all descending nodes. However, we choose to only store a single path ID with which to represent an element from our database, since we can easily select the entire contents when needed, with the help of our implementation of the XPath axis `descendant-or-self::`.

In order to create the correct tuple stream, we must start by evaluating the expression "`//book`", associated with the `$b` variable. For each iteration of the three books we must then branch out and iterate over the evaluation of the expression in the next `for` clause. Each evaluation of the expression `$b/author` differs for each book, in both value and number of results.

For the purpose of generating the initial tuple stream, we propose the algorithm in Listing 3. We assume that each `for` and `let` clause binds a single variable to an expression. These expressions are accessed through an array `$expr[]` of a datatype that holds the variable name, the associated expression, and a type flag indicating whether the binding is a `for` or a `let` clause. The array is indexed by the order in which the variable bindings appear in the FLWOR expression. In practise, this access to the expressions could be implemented in a number of ways.

The `filltable()`function takes a `$scope_tuple` argument. This argument is a tuple containing the variable bindings that may be used in processing the current expression, thus the scope. The other argument of the function, `$expr_num`, is the means by which we move on to the next `for/let` clause in the `$expr[]` array each time we perform another recursive function call.

Furthermore, our algorithm makes use of a function which we have named `binder()`. This `binder()` function is the mechanism that binds the result of an expression to a variable by creating the a table which represents one variable. Its basic operation is to run our XPath evaluator on the expression it recieves as an argument and enter the results into a single-column table. The result returned by the `binder()` function differs for `for` and `let` clauses. `for`-bound variables are returned as a table containing one tuple for each item in the evaluation of its associated expression. `let`-bound variables are returned as a table containing just a single tuple which holds the entire evaluation of the associated expression. `binder()` also takes a `$scope_tuple` argument. If the expression being evaluated requires the value of a previously bound variable, then `binder()` retrieves this value from the `$scope_tuple`.

In the body of the `filltable()` function, we create a table designated `$return_table`, which is used to store the results fetched from the recursive function calls as they are made. This table has one column for every variable binding stored in `$expr[]`within the interval `[$expr_num...`*`<last $expr_num>`*`]`. We then call `binder()` with the current expression and the scope tuple. After receiving a result from `binder()`, we make recursive calls to `filltable()` for each tuple in this result, in order to join our scope tuple with the results of the next `for/let` clause, and add this to the `$return_table`. Each tuple resulting from `binder()` is also used as the `$scope_tuple` in these recursive calls to `filltable()`. At the end of the execution of a `filltable()` call, we return the resulting table, which contains the tuples that the calling function needs to join with its own scope tuple.

The `where` *clause*

After creating the initial tuple stream as described above, the `where` clause may be applied. The `where` clause contains a conditional statement that must evaluate to true in order for the tuple in question to survive. It is necessary to perform additional XPath or XQuery processing beforehand, in order to fully resolve a clause such as "`where $book[author="John Doe"]`", or any other expression that includes additional path steps or even full XQuery statements. For instance, in this example the current iteration of `$book` could contain the path ID "`1/2/42`". We would then have to evaluate "`child::author/child::text()`" with the node "`1/2/42`" as the context node in order to obtain a value to compare to the string "`John Doe`".

The `order by` *clause*

At this point we will be ready to apply ordering to the stream. In this step, we might also need to perform additional XPath evaluations in order to retrieve string values to sort by. For example, in the case of an `order by` clause such as "`order by $b/title`", we would need to evaluate

```
1 filltable($scope_tuple , $expr_num){
2   create $return_table with one column for every binding in $expr[$exprnum...<last $exprnum>]
3   $result := binder($scope_tuple , $expr[$expr_num])
4   for all $tuple in $result
5     insert into $return_table the values: $scope_tuple joined with filltable($tuple , $expr_num + 1)
6   return $return_table
7 }
```

Listing 3: Our algorithm for generating the tuple stream.

"`child::title/child::text()`" on each node in the tuple stream and order the stream by these results.

The `return` *clause*

Upon reaching the `return` statement, the tuple stream is ready for output. The return statement iterates over each of the records in the final tuple stream and returns the desired output once for each of these. In this step, also, we are still required to perform lookups. If, for instance, the result stored in a variable `$book` is the path ID of a `book` element and this variable is included in the `return` statement, then the output string needs to be the entire contents of the element. Within our implementation, we would perform a `descendant-or-self::node()` operation with the book node as the context node in order to receive a complete set of nodes that we can send to our round-tripper. The round-tripper would then return the appropriate XML fragment to insert in place of the variable in each iteration of the `return` statement.

5. EVALUATION

In this section we evaluate our implementation in general, and make comparisons to other studies, with different approaches.

5.1 Database schemas

Our approach to creating the relational schema is quite similar to that in [9] called *Edge*. In the original Edge, path IDs are stored through references to parent nodes, whereas we store a full path ID for each tuple in the database. Using the path ID encoding enables us to identify the path and the depth of a given node immediately, whereas references to parent nodes means that you have to calculate your path ID for a given node. Also, Edge uses only one table where we use two. This means that Edge has to save null values since the Edge approach saves two node types with different relational attributes in the same table. For example, when saving an attribute, one would store both the name and the value of the attribute, whereas when storing an element you would only store the name, since an element does not have a value. In the case of an element, the value field would simply contain a null value. We do not have to do this since we have different tables for attributes and elements. To decide which one is the best approach you would have to take space versus time complexity into consideration; it is a matter of deciding between storing null valued fields, versus searching in two tables.

Another approach for relational schema layout, is the *Shared* approach [15]. It makes use of XML schemas to define the relational schema. A table will be created for the root node, and XML attributes and elements which only occur once will be the attributes of the database. If the element can contain other elements which can occur more than once, a new table wil be created for that element and so on. This can result in many tables when dealing with complex XML schemas, which might result in overhead when trying to locate a table. It does, though, give a better conceptually understandable representation of the data, than just using a single table, which might be an advantage when talking about using XML views [15].

Our approach of only using two tables makes our model more general and simply structured, than the Shared. Furthermore our tables are easily located, and since all searching is done within only these two tables, we avoid some database overhead of sorting out references. On the other hand searching through one large table as in our case, is more time consuming than searching one of the smaller tables in the shared approach. But as mentioned the smaller table needs to be located first, so again it is a matter of time and space complexity.

5.2 Handling several XML documents

In our approach it is not a problem to shred several documents and insert them into the database. Because of the encoding we use it is simply a matter of changing the root number of the path ID to a consecutive number. This method cannot be used in the Shared approach since tables are dependent on a XML schema definition, which means that a new set of tables will have to be made for each document. In this regard, Shared less flexible than our solution.

5.3 Dewey path encoding

In Section 3.1.2 we describe a naive and an optimal solution to ordering Dewey paths. The former involved prefixing path steps with zeroes and performing string sorts and the latter proposed a more complex solution. Due to time constraints we have chosen only to implement the naive solution.

5.4 XPath and XQuery implementation issues

In our implementation of XPath we have chosen to put all the functionality in the database. The obvious advantage is that we only have to deal with one language. But also it is only necessary to administrate one implementation. This is beneficial if one considers using client software written in different languages, and eventually for different platforms. Placing the functionality in the database also minimizes net-

work traffic, and exploits the database optimization capabilities.

Eventhough we have only given an outline for the XQuery implementation, we belive the same approach should be followed as for XPath, i.e. put the functionality in the database. This solution seems ideal for the same reasons as for XPath, both also there is a close relation between e.g. the way we solve nested for-clauses, and join mechanisms in an RDBMS.

6. CONCLUSION

In this article we have presented methods for storing and querying XML data in a relational database. We have shown a method for storing XML data using a simple database schema, and how XML data can be shredded to the database, using a specific Dewey encoding. We have also shown how XPath axis steps can be implemented, giving specific SQL queries for how routines can be created in the database. Further we have given an outline for how the XQuery FLWOR construct can be implemented. We have also given the algorithms for reconstructing an XML document, or a fragment of it.

We have argued that our implementation is simple and general, by comparing it to related work. We believe that our method has some advantages compared to these other studies in the context of simplicity and generality. Though many aspects have not been taken into consideration. Future work could include comparison with other studies regarding space and time complexity, using statistics for a more exact evaluation of different methods.

7. ACKNOWLEDGEMENTS

We thank Albrecht Schmidt, for supervising the process of development and the writing of this article. We also thank our fellow students Christian Andersen, Tim Boesen and Dennis Kjærulff for valuable discussions on the subject of XML and related theory. All mentioned individuals are, at the time of writing, affiliated to the Department of Computer Science at Aalborg University, Denmark.

8. RELATED WORK

Shredding, querying, and round-tripping XML data has been the topic of various earlier articles. [13, 15, 16] deal with shredding and round-tripping representing XML files as trees in the database. Additionally [16] makes use of a "flat" database representation. [9] uses a tree-representation in the database, but provides a "flat" view for querying. [19] focuses on order encoding methods ("Global Order", "Local Order", and "Dewey Order") that promise to keep the ordering of XML documents in an unordered database. [11] describes the mapping of DTDs onto object-relational database systems. The main topics of [13, 14] are querying, the latter using the shredding strategies "Shared Inline" and "Edge" from other articles as approaches for case studies. [7] also deals with querying but outlines an implementation of an XQuery and XPath compiler, proposing the use of equi-joins for implementing the iterative, for-bound variables.

9. REFERENCES

[1] *About SAX*. `http://sax.sourceforge.net/`.

[2] Joan Aliprand, Julie Allen, Joe Becker, Mark Davis, Michael Everson, Asmus Freytag, John Jenkins, Mike Ksar, Rick McGowan, Eric Muller, Lisa Moore, Michel Suignard, and Ken Whistler, editors. *The Unicode Standard – Version 4.0*, chapter 3.9, page 77. The Unicode Consortium, `http://www.unicode.org/versions/Unicode4.0.0/`, August 2003.

[3] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. *XQuery 1.0: An XML Query Language*. W3C, `http://www.w3.org/TR/xquery`, November 2003.

[4] John Boyer. *Canonical XML 1.0 Recommendation*. W3C, `http://www.w3.org/TR/xml-c14n`, March 2001.

[5] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eva Maler, and François Yergau. *Extensible Markup Language (XML) 1.0*. W3C, `http://www.w3.org/TR/REC-xml/`, 3rd edition, Feburary 2004.

[6] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eva Maler, and François Yergau. *Extensible Markup Language (XML) 1.0*. W3C, `http://www.w3.org/TR/REC-xml/`, 3rd edition, Feburary 2004. Section 3.1.

[7] Byron Choi, Mary Fernández, and Jérôme Siméon. The XQuery Formal Semantics: A Foundation for Implementation and Optimization. Technical report, University of Pennsylvania, 2002.

[8] James Clark and Steve DeRose. *XML Path Language (XPath)*. W3C, `http://www.w3.org/TR/xpath`, November 1999.

[9] Daniela Florescu and Donald Kossmann. Storing and Querying XML Data using an RDBMS. *IEEE Data Engeneering Bulletin*, 22(3):27–34, 1999.

[10] Apache Software Foundation. Xerces. `http://xml.apache.org/`.

[11] Meike Klettke and Holger Meyer. XML and Object-Relational Database Systems: Enhancing Structural Mappings Based on Statistics. In *The World Wide Web and Databases: Third International Workshop WebDB 2000*, volume 1997, pages 151–170. Springer-Verlag Heidelberg, 2001.

[12] Edward M. McCreight Rudolf Bayer. Organization and maintenance of large ordered indices. In *Acta Informatica*, volume 1, pages 173–189, 1972.

[13] Albrecht Schmidt, Martin Kersten, Menzo Windhouwer, and Florian Waas. Efficient Relational Storage and Retrieval of XML Documents. In *The World Wide Web and Databases: Third International Workshop WebDB 2000*, volume 1997, pages 137–150. Springer-Verlag Heidelberg, 2001.

[14] Jayavel Shanmugasundaram, Eugene Shekita, Jerry Kieman, Rajasekar Krishnamurthy, Efstratios Viglas, Jeffrey Naughton, and Igor Tatarinov. A General Technique for Querying XML Documents using a Relational Database System. *ACM SIGMOD*, 30(3):20–26, September 2001.

[15] Jayavel Shanmugasundaram, Kristin Tufte, Gang He, Chun Zhang, David DeWitt, and Jeffrey Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *25th Very Large Data Base Endowment Conference*, 1999.

[16] Takeyuki Shimura, Masatoshi Yoshikawa, and Shunsuke Uemura. Storage and Retrieval of XML Documents Using Object-Relational Databases. In *Database and Expert Systems Applications: 10th International Conference, DEXA '99*, volume 1677, page 206. Springer-Verlag Heidelberg, 1999.

[17] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*, chapter 4-7. McGraw-Hill, 4th edition, 2002.

[18] Kimbro Staken. Introduction to Native XML Databases, October 2001. `http://www.xml.com/pub/a/2001/10/31/nativexmldb.html`.

[19] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *2002 ACM SIGMOD International Conference on Management of Data*, pages 204–215. ACM Press, 2002.

[20] W3C, `http://www.w3.org/DOM/`. *Document Object Model (DOM)*.

[21] W3C. *HyperText Markup Language Homepage*. W3C, `http://www.w3.org/MarkUp`.

[22] WhatIs.com. `http://whatis.com/`. Search for "roundtripping".

References containing URLs are valid as of May 28, 2004.