MXML Storage and the Problem of Manipulation of Context

Nikolaos Fousteris[1], Manolis Gergatsoulis[1], and Yannis Stavrakas[2]

[1] Database & Information Systems Group (DBIS),
Laboratory on Digital Libraries and Electronic Publishing,
Department of Archives and Library Science, Ionian University,
Ioannou Theotoki 72, 49100 Corfu, Greece.
{nfouster,manolis}@ionio.gr,
[2] Institute for the Management of Information Systems (IMIS),
R. C. Athena,
G. Mpakou 17, 11524, Athens, Greece.
yannis@imis.athena-innovation.gr

Abstract. The problem of storing and querying XML data using relational databases has been considered a lot and many techniques have been developed. MXML is an extension of XML suitable for representing data that assume different facets, having different value and structure under different contexts, which are determined by assigning values to a number of dimensions. In this paper, we explore techniques for storing MXML documents in relational databases, based on techniques previously proposed for conventional XML documents. Essential characteristics of the proposed techniques are the capabilities a) to reconstruct the original MXML document from its relational representation and b) to express MXML context-aware queries in SQL.

1 Introduction

The problem of storing XML data in relational databases has been intensively investigated [4, 10, 11, 13] during the past 10 years. The objective is to use an RDBMS in order to store and query XML data. First, a relational schema is chosen for storing the XML data, and then XML queries, produced by applications, are translated to SQL for evaluation. After the execution of SQL queries, the results are translated back to XML and returned to the application.

Multidimensional XML (MXML) is an extension of XML which allows context specifiers to qualify element and attribute values, and specify the contexts under which the document components have meaning. MXML is therefore suitable for representing data that assume different facets, having different value or structure, under different contexts. Contexts are specified by giving values to one or more user defined dimensions. In MXML, dimensions may be applied to elements and attributes (their values depend on the dimensions). An alternative solution would be to create a different XML document for every possible combination, but such an approach involves excessive duplication of information.

In this paper, we present two approaches for storing MXML in relational databases, based on XML storage approaches. We use MXML-graphs, which are graphs using appropriate types of nodes and edges, to represent MXML documents. In the first (naive) approach, a single relational table is used to store all information about the nodes and edges of the MXML-graph. Although simple, this approach presents some drawbacks, like the large number of expensive self-joins when evaluating queries. In the second approach we use several tables, each of them storing a different type of nodes of the MXML-graph. In this way the size of the tables involved in joins is reduced and consequently the efficiency of query evaluation is enhanced. Both approaches use additional tables to represent context in a way that it can be used and manipulated by SQL queries. Additionally to MXML storage, we propose techniques for context manipulation, as context is one of the major characteristics of MXML.

2 Preliminaries

2.1 Mutidimensional XML

In MXML, data assume different facets, having different value or structure, under different contexts according to a number of *dimensions* which may be applied to elements and attributes [7, 8]. The notion of "world" is fundamental in MXML. A world represents an environment under which data obtain a meaning. A *world* is determined by assigning to every dimension a single value, taken from the domain of the dimension. In MXML we use syntactic constructs called *context specifiers* that specify sets of worlds by imposing constraints on the values that dimensions can take. The elements/attributes that have different facets under different contexts are called *multidimensional elements/attributes*. Each multidimensional element/attribute contains one or more facets, called *context elements/attributes*, accompanied with the corresponding context specifier which denotes the set of worlds under which this facet is the holding facet of the element/attribute. The syntax of MXML is shown in Example 1, where a MXML document containing information about a book is presented.

Example 1. The MXML document shown below represents a book in a book store. Two dimensions are used namely `edition` whose domain is {`greek, english`}, and `customer_type` whose domain is {`student, library, teacher`}.

```
<book isbn=[edition=english]"0-13-110362-8"[/]
           [edition=greek]"0-13-110370-9"[/]>
  <title>The C programming language</title>
  <authors>
     <author>Brian W. Kernighan</author>
     <author>Dennis M. Ritchie</author>
  </authors>
  <@publisher>
     [edition = english] <publisher>Prentice Hall</publisher>[/]
     [edition = greek] <publisher>Klidarithmos</publisher>[/]
  </@publisher>
  <@translator>
    [edition = greek] <translator>Thomas Moraitis</translator>[/]
  </@translator>
  <@price>
    [edition=english]<price>15</price>[/]
    [edition=greek,customer_type in {student, teacher}]<price>9</price>[/]
    [edition=greek,customer_type=library]<price>12</price>[/]
  </@price>
  <@cover>
    [edition=english]<cover><material>leather</material></cover>[/]
    [edition=greek]
      <cover>
        <material>paper</material >
        <@picture>
          [customer_type=student]<picture>student.bmp</picture>[/]
          [customer_type=library]<picture>library.bmp</picture>[/]
        </@picture>
       </cover>
     [/]
  </@cover>
</book>
```

Notice that multidimensional elements (see for example the element `price`) are the elements whose name is preceded by the symbol `@` while the corresponding context elements have the same element name but without the symbol `@`.

A MXML document can be considered as a compact representation of a set of (conventional) XML documents, each of them holding under a specific world. For the extraction of XML documents holding under specific worlds the interested reader may refer to [7] where a related process called *reduction* is presented.

2.2 Storing XML data in relational databases

Many researchers have investigated how an RDBMS can be used to store and query XML data. Work has also been directed towards the storage of temporal extensions of XML [16, 1, 2]. The techniques proposed for XML storage can be divided in two categories, depending on the presence or absence of a schema:

1. *Schema-Based XML Storage techniques*: the objective here is to find a relational schema for storing a XML document, guided by the structure of a schema for that document [9, 13, 5, 15, 10, 3, 11].
2. *Schema-Oblivious XML Storage techniques*: the objective is to find a relational schema for storing XML documents independent of the presence or absence of a schema [13, 5, 15, 17, 10, 6, 4].

The approaches that we propose in this paper do not take schema information into account, and therefore belong to the Schema-Oblivious category.

3 Properties of MXML documents

3.1 A graphical model for MXML

In this section we present a graphical model for MXML called *MXML-graph*. The proposed model is node-based and each node is characterized by a unique "id". In MXML-graph, except from a special node called *root node*, there are the following node types: *multidimensional element nodes*, *context element nodes*, *multidimensional attribute nodes*, *context attribute nodes*, and *value nodes*. The *context element nodes*, *context attribute nodes*, and *value nodes* correspond to the element nodes, attribute nodes and value nodes in a conventional XML graph. Each multidimensional/context element node is labelled with the corresponding element name, while each multidimensional/context attribute node is labelled with the corresponding attribute name. As in conventional XML, value nodes are leaf nodes and carry the corresponding value. The facets (context element/attribute nodes) of a multidimensional node are connected to that node by edges labelled with context specifiers denoting the conditions under which each facet holds. These edges are called *element/attribute context edges* respectively. Context elements/attributes are connected to their child elements/attribute or value nodes by edges called *element/attribute/value edges* respectively. Finally, the context attributes of type IDREF(S) are connected to the element nodes that they point to by edges called *attribute reference edges*.

Example 2. In Fig. 1, we see the representation of the MXML document of Ex-

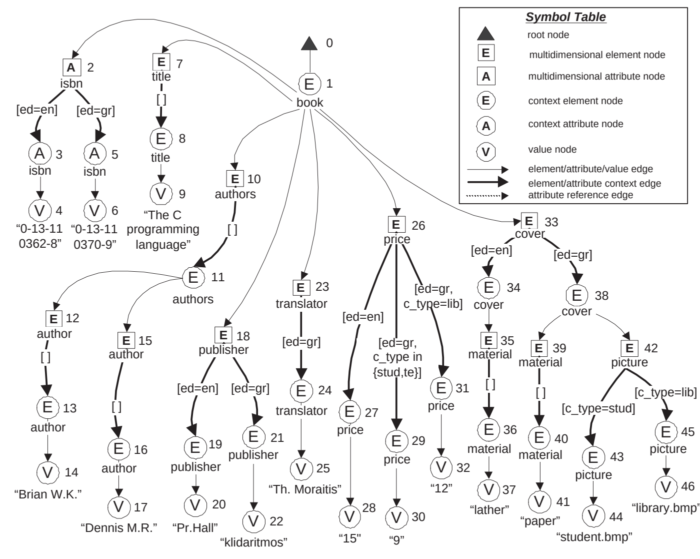

Fig. 1. Graphical representation of MXML (MXML tree)

ample 1 as a MXML-graph. Note that some additional multidimensional nodes (e.g. nodes 7 and 10) have been added to ensure that the types of the edges alternate consistently in every path of the graph. This does not affect the information contained in the document, but facilitates the navigation in the graph and the formulation of queries. For saving space, in Fig. 1 we use obvious abbreviations for dimension names and values that appear in the MXML document.

3.2 Properties of contexts

Context specifiers qualifying element/attribute context edges give the *explicit contexts* of the nodes to which these edges lead. The explicit context of all the other nodes of the MXML-graph is considered to be the *universal context* [ ], denoting the set of all possible worlds. The explicit context can be considered as the true context only within the boundaries of a single multidimensional element/attribute. When elements and attributes are combined to form a MXML document, the explicit context of each element/attribute does not alone determine the worlds under which that element/attribute holds, since when an element/attribute e_2 is part of another element e_1, then e_2 have substance only under the worlds that e_1 has substance. This can be conceived as if the context under which e_1 holds is inherited to e_2. The context propagated in that way is combined with (constraint by) the explicit context of a node to give the *inherited context* for that node. Formally, the inherited context $ic(q)$ of a node q is defined as $ic(q) = ic(p) \cap^c ec(q)$, where $ic(p)$ is the inherited context of its parent node p. $\cap^c$ is an operator called *context intersection* defined in [12] which combines two context specifiers and computes a new context specifier which represents the intersection of the worlds specified by the original context specifiers. The evaluation of the inherited context starts from the root of the MXML-graph. By definition, the inherited context of the root of the graph is the universal context [ ]. Note that contexts are not inherited through attribute reference edges.

As in conventional XML, the leaf nodes of MXML-graphs must be value nodes. The *inherited context coverage* of a node further constraints its inherited context, so as to contain only the worlds under which the node has access to some value node. This property is important for navigation and querying, but also for the reduction process [7]. The inherited context coverage $icc(n)$ of a node n is defined as follows: if n is a leaf node then $icc(n) = ic(n)$; otherwise $icc(n) = icc(n_1) \cup^c icc(n_2) \cup^c ... \cup^c icc(n_k)$, where $n_1, \dots, n_k$ are the child element nodes of n. $\cup^c$ is an operator called *context union* defined in [12] which combines two context specifiers and computes a new one which represents the union of the worlds specified by the original context specifiers. The inherited context coverage gives the true context of a node in a MXML-graph.

4 Storing MXML in relational databases

In this section we present two approaches for storing MXML documents using relational databases.

4.1 Naive Approach

The first approach, called *naive approach*, uses a single table (*Node Table*), to store all information contained in a MXML document. Node Table contains all the information which is necessary to reconstruct the MXML document(graph). Each row of the table represents a MXML node. The attributes of Node Table are: `node_id` stores the id of the node, `parent_id` stores the id of the parent node, `ordinal` stores a number denoting the order of the node among its siblings, `tag` stores the label (tag) of the node or NULL (denoted by "-") if it is a value node, `value` stores the value of the node if it is a value node or NULL otherwise, `type` stores a code denoting the node type (CE for context element, CA for context attribute, ME for multidimensional element, MA for multidimensional attribute, and VN for value node), and `explicit_context` stores the explicit context of the node (as a string). Noted that the explicit context is kept here for completeness, and does not serve any retrieval purposes. In the following we will see how the correspondence of nodes to the worlds under which they hold is encoded.

Example 3. Fig. 2 shows how the MXML Graph of Fig. 1 is stored in the Node Table. Some of the nodes have been omitted, denoted by "....", for brevity.

Node Table						
node_id	parent_id	ordinal	tag	value	type	explicit_context
1	0	1	book	-	CE	-
2	1	1	isbn	-	MA	-
3	2	1	isbn	-	CA	[ed=en]
4	3	1	-	0-13-110362-8	VN	-
5	2	2	isbn	-	CA	[ed=gr]
6	5	1	-	0-13-110370-9	VN	-
7	1	2	title	-	ME	-
8	7	1	title	-	CE	[ ]
9	8	1	-	The C progr. lang.	VN	-
....	....	....	....	....	....	....
43	42	1	picture	-	CE	[c_type=stud]
....	....	....	....	....	....	....

Fig. 2. Storing the MXML-graph of Fig. 1 in a Node Table.

4.2 Limitations of the Naive Approach

The naive approach is straightforward, but it has some drawbacks mainly because of the use of a single table. As the different types of nodes are stored in the table, many NULL values appear in the fields `explicit_context`, `tag`, and `value`. Those NULL values could be avoided if we used different tables for different node types. Moreover, as we showed in Subsection 4.1, queries on MXML documents involve a large number of self-joins of the Node Table, which is anticipated to be a very long table since it contains the whole tree. Splitting the Node Table would reduce the size of the tables involved in joins, and enhance the overall performance of queries. Finally, notice that the context representation scheme we introduced leads to a number of joins in the nested query. Probably a better scheme could be introduced that reduces the number of joins.

4.3 A Better Approach

In the *Type Approach* presented here, MXML nodes are divided into groups according to their type. Each group is stored in a separate table named after the type of the nodes. In particular *ME Table* stores multidimensional element nodes, *CE Table* stores context element nodes, *MA Table* stores multidimensional attribute nodes, *CA Table* stores context attribute nodes, and *Value Table* stores value nodes. The schema of these tables is shown in Fig. 3. Each row in these

ME Table			
node_id	parent_id	ordinal	tag
7	1	2	title
10	1	3	authors
....	....	....	....

CE Table				
node_id	parent_id	ordinal	tag	explicit_ context
1	0	1	book	-
8	7	1	title	[ ]
....	....	....	....	....
19	18	1	publisher	[ed=en]
21	18	2	publisher	[ed=gr]
....	....	....	....	....

MA Table			
node_id	parent_id	ordinal	tag
2	1	1	isbn

CA Table				
node_d	parent_id	ordinal	tag	explicit_context
3	2	1	isbn	[ed=en]
5	2	2	isbn	[ed=gr]

Value Table		
node_id	parent_id	value
4	3	0-13-110362-8
6	5	0-13-110362-9
9	8	The C programming language
....	....	....

Fig. 3. The Type tables.

tables represents a MXML node. The attributes in the tables have the same meaning as the respective attributes of the Node Table. Using this approach we tackle some of the problems identified in the previous section. Namely, we eliminate NULL values and irrelevant attributes, while at the same time we reduce the size of the tables involved in joins when navigating the MXML-Graph.

5 Context Representation

In this section we present techniques that help us to store the context in such a way so as to facilitate the formulation of context-aware queries. Two approaches, for storing context in a Relational Database, are presented. The first, is a naive representation and the second one is called the Ordered-Based representation.

5.1 Naive Context Representation

For the Naive Context Representation technique, we introduce three additional tables, as shown in Fig. 4. The *Possible Worlds Table* which assigns a unique ID (attribute `word_id`) to each possible combination of dimension values. Each dimension in the MXML document has a corresponding attribute in this table. The *Explicit Context Table* keeps the correspondence of each node with the worlds represented by its explicit context. Finally, the *Inherited Coverage Table* keeps the correspondence of each node with the worlds represented by its inherited context coverage.

Example 4. Fig. 4, depicts (parts of) the Possible Worlds Table, the Explicit Context Table, and the Inherited Coverage Table obtained by encoding the context information appearing in the MXML-graph of Fig. 1. For example, the

Possible Worlds Table		
world_id	edition	customer_type
1	gr	stud
2	gr	lib
3	gr	te
4	en	stud
5	en	lib
6	en	te

Explicit Context Table	
node_id	world_id
1	1
1	2
1	3
1	4
1	5
1	6
....	....
5	1
5	2
5	3
6	1
6	2
6	3
6	4
6	5
6	6
....	....

Inherited Coverage Table	
node_id	world_id
1	1
1	2
1	3
1	4
1	5
1	6
....	....
5	1
5	2
5	3
6	1
6	2
6	3
....	....

Fig. 4. Mapping MXML nodes to worlds.

inherited context coverage of the node with `node_id=6` includes the worlds:

$w_1 = \{$`(edition, greek)`, `(customer_type, student)`$\}$,
$w_2 = \{$`(edition, greek)`, `(customer_type, library)`$\}$ and
$w_3 = \{$`(edition, greek)`, `(customer_type, teacher)`$\}$

This is encoded in the Inherited Coverage Table as three rows with `node_id=6` and the world ids 1, 2 and 3. In the Explicit Context Table the same node corresponds to all possible worlds (ids 1, 2, 3, 4, 5 and 6).

5.2 Ordered-Based Context Representation

According to the Ordered-Based Context Representation technique, we propose a scheme that reduces the size of tables and the number of joins in context-driven queries. The basic idea of this technique is that we achieve the total ordering of all possible worlds based on a) a total ordering of dimensions and b) a total ordering of dimension possible values. So, for k dimensions with each dimension i having m_i possible values, we may have $n = m_1 * m_2 * \ldots * m_k$ possible ordered worlds. Each of these worlds is assigned a unique integer value between 1 and n.

Example 5. In Fig. 5, we present how it is possible to order all possible worlds according to the dimensions and the dimension values of Example 1. In order

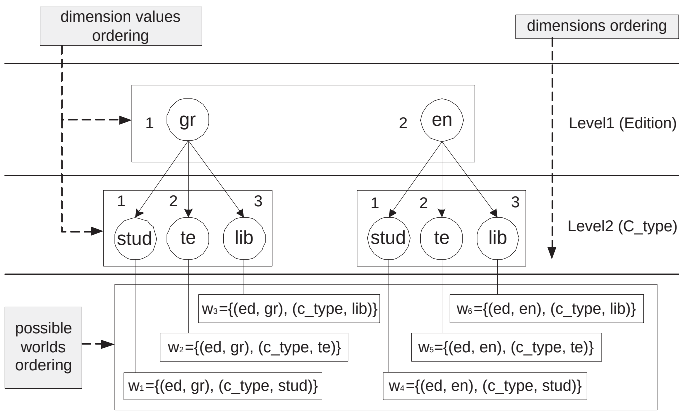
dimension values ordering
dimensions ordering
1
gr
2
en
Level1 (Edition)
1
2
3
stud
te
lib
1
2
3
stud
te
lib
Level2 (C_type)
possible worlds ordering
w₃={(ed, gr), (c_type, lib)}
w₂={(ed, gr), (c_type, te)}
w₁={(ed, gr), (c_type, stud)}
w₆={(ed, en), (c_type, lib)}
w₅={(ed, en), (c_type, te)}
w₄={(ed, en), (c_type, stud)}

Fig. 5. Possible Worlds Ordering

to show this ordering, we use a forest of trees. As we can see, each dimension of the MXML document corresponds to a level in the forest. The ordering of these levels represents the ordering of dimensions. Also, for each level we can see the ordering of all possible values of the related dimension, under each node of the previous level. Each possible world can be produced by traversing a path from a root node of the forest to a leaf node of the corresponding tree. Finally, the order of the forest's leaves represents the total ordering of all possible worlds assigning a unique integer to each world $(w_1, w_2, \ldots, w_6)$.

Assuming that all possible worlds of a MXML document are totaly ordered, we define a vector of binary digits called World Vector.

Definition 1. *Given a total ordering of worlds $W = (w_1, w_2, \ldots, w_n)$, where n is the number of possible worlds, we define as $V(c) = (a_1, a_2, \ldots, a_n)$ the World Vector of a context specifier c, where a_i with $i = 1, 2, \ldots, n$, is a one bit value containing* 1 *if the world w_i is between the worlds represented by c or* 0 *if w_i is not included in the worlds represented by c.*

In Fig. 6 we can see how in general we can store dimensions' information to the Relational Database. One table (Table D) is used for storing ordered dimensions and one separate table D_i with $i = 1, 2, \ldots, k$ is used for storing the ordered values $d_{i,j}$ with $j = 1, 2, \ldots, m_i$ and m_i is the number of the different values of dimension D_i.

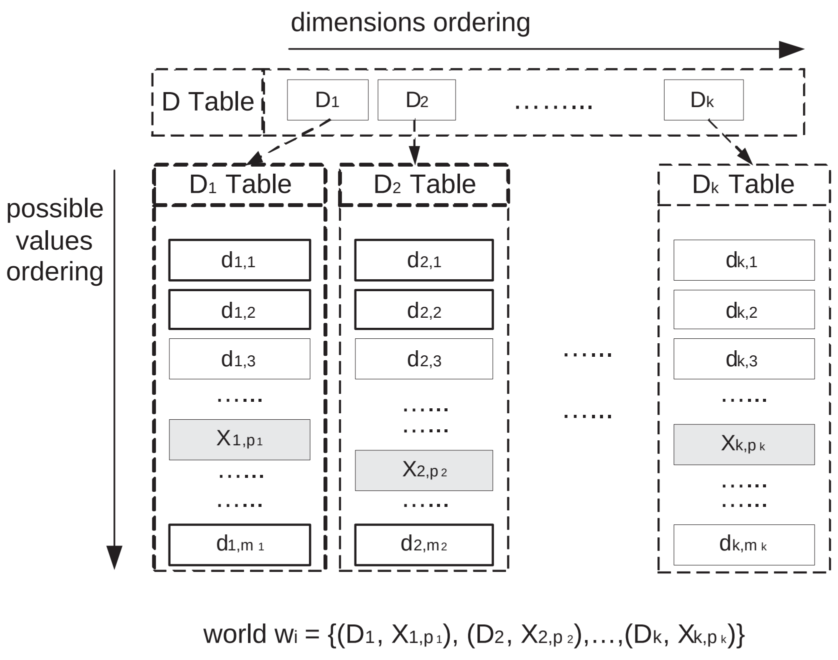

Fig. 6. Ordered-Based Representation in Relational Schema

5.2.1 Finding the position of a world in a World Vector A problem which arise when using the Ordered-Based Representation to represent worlds, is the problem of defining the position corresponding to a specific world in a world vector. Assuming that a context specifier contains the world w_i, shown in Fig. 6, we can find the bit-position i corresponding to this world in the world

D Table	
dimension_id	dimention_name
1	edition
2	customer_type

D_1 Table	
value_id	value
1	greek
2	english

D_2 Table	
value_id	value
1	student
2	teacher
3	library

Inherited Coverage Table	
node_id	world_vector
1	111111
2	111111
3	000111
4	000111
5	111000
6	111000
....	....

Explicit Context Table	
node_id	world_vector
1	111111
2	111111
3	000111
....	....
31	001000
....	....
43	100100
....	....

Fig. 7. Context Tables.

vector of the context specifier, using the following formula:

$$i = p_k + \sum_{j=2}^{k} [(p_{j-1} - 1) * (\prod_{w=j}^{k} m_w)]$$

Example 6. Fig. 7, depicts (parts of) the Explicit Context Table, and the Inherited Coverage Table obtained by encoding the context information appearing in the MXML-graph of Fig. 1. Also, we can see the contents of the tables D, D_1 and D_2 containing the ordering information for all possible worlds. For example, the explicit context of the node with `node_id=3` includes the worlds:

$w_1 = \{$`(edition, english)`, `(customer_type, student)`$\}$,
$w_2 = \{$`(edition, english)`, `(customer_type`, `teacher)`$\}$ and
$w_3 = \{$`(edition, english)`, `(customer_type`, `library)`$\}$

According to the ordering of Fig. 5, the bit-positions of these worlds in the world vector are 4, 5 and 6 respectively. As a result, the explicit context specifier of the node is encoded in the Explicit Context Table as one row with `node_id=3` and the world vector 000111.

5.2.2 Finding the world corresponding to a bit in a World Vector

The opposite problem of finding the position of a world in a world vector is the problem of finding which world corresponds to a bit-position i of a world vector. In order to achieve this, we can use the algorithm represented by the flowchart shown in Fig. 8, using the notation of Fig. 5. The algorithm of Fig. 8 takes as input the i position of a world in a world vector. The output of the algorithm is a sequence of numbers $(p_1, p_2, \ldots, p_k)$. Each number p_i represents the position of a value among the ordered values of dimension D_i. Using this position, it is

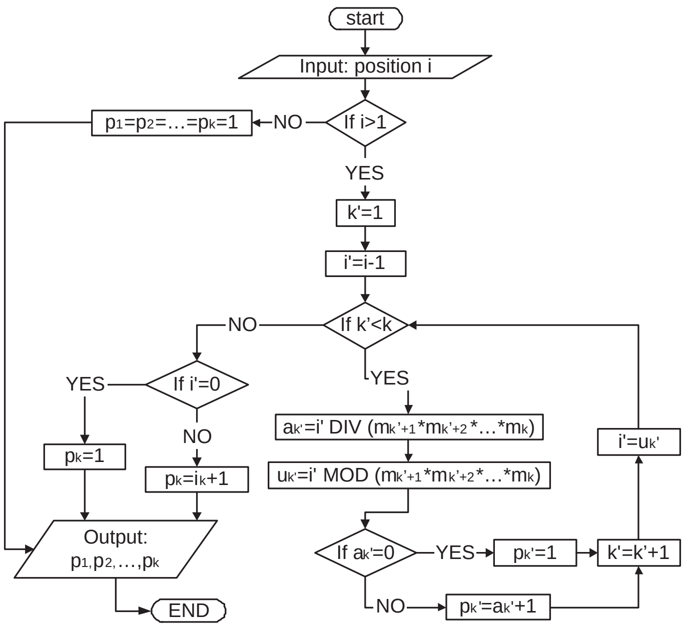

Fig. 8. Converting bit-position i of world vector to world

possible to find the value X_{i,p_i} of the dimension D_i from the appropriate table D_i of Fig. 5. A The set of pairs (D_i, X_{i,p_i}) represents the resulting world.

6 Querying MXML with Multidimensional XPath

In this section we present *Multidimensional XPath* (MXPath) as an extension of XPath used to navigate through MXML-graphs. In addition to the conventional XPath functionality, MXPath uses the inherited context coverage and the explicit context of MXML in order to select nodes in the MXML document. Similarly to XPath, MXPath uses *path expressions* as a sequence of steps to get from one MXML node to another node, or set of nodes.

In a MXPath, selection criteria concerning the explicit context are expressed through *explicit context qualifiers*. Selection criteria concerning the inherited context coverage are expressed through the *inherited context coverage qualifier*, which is placed at the beginning of the expression.

6.1 MXPath Syntax

An *MXPath expression* contains an *inherited context coverage qualifier* (or *icc qualifier* for short) followed by the *MXPath expression body*. The inherited context coverage qualifier is placed at the beginning of the expression and filters the resulting nodes according to their inherited context coverage. The syntax of an MXPath expression is:

`[inherited_context_coverage_qualifier],MXPath_expression_body`

An MXPath expression may return either multidimensional nodes or context nodes. In what follows we brake down MXPath expressions, and specify each part separately.

6.1.1 Inherited context coverage qualifier

The syntax of the *inherited context coverage qualifier* is:

`icc() comparison_op context_specifier_expression`

where `comparison_op` is one of the operators `=`, `!=`, `<`, `>`, `<=`, or `>=`. Note that it is easy to prove that for the inherited context coverages of the nodes in a path $r, n_1, \ldots, n_k$, from the root r of the MXML tree to a node n_k, it holds that $icc(n_k) \subseteq icc(n_{k-1}) \subseteq \ldots \subseteq icc(r)$. Thus $icc(n_k)$ denotes the worlds under which the complete path holds. The function `icc()` returns the icc of the current node, and, consequently of the currently evaluated path in MXML. This icc is then compared against the *context_specifier*, according to the comparison operator. The operator `=` tests for equality, `<` tests for proper subset, `>` for proper superset, etc. Note that it is actually the sets of worlds represented by the contexts that are compared. In case the comparison returns *false*, the current path is rejected and not considered further. If the inherited context coverage qualifier is omitted in an MXPath expression, the default is implied: `icc() >= "-"`, which evaluates always to *true*.

6.1.2 MXPath expression body

MXPath expression body corresponds to (conventional) XPath expressions. As in XPath, in MXPath we also have two types of expression bodies, namely the *absolute* and the *relative*. An absolute MXPath expression body is a relative one preceded by the symbol "/" which denotes the root of the MXML tree. `MXPath_expression_body` is composed by one of more *MXPath steps* separated by "/". Thus, the syntax of a relative `MXPath_expression_body` is of the form:

`MXPath_step_1/MXPath_step_2/.../MXPath_step_n`

6.1.3 MXPath steps

There are two types of MXPath steps, namely, the *Context MXPath steps* which return context nodes, and the *Multidimensional MXPath steps* which return multidimensional nodes. The syntax of a Context MXPath step is as follows:

`axis::node_test[pred_1][pred_2]...[pred_n]`

while the syntax of a Multidimensional MXPath step is as follows:

`axis->node_test[pred_1][pred_2]...[pred_n]`

Notice that, both types of MXPath steps contain an *axis*, a *node test* and zero or more *predicates*. The only difference is that in a context MXPath step the axis is followed by the symbol "`::`" which denotes that the step evaluates to context nodes, while in a Multidimensional MXPath step axis is followed by the symbol "`->`" which denotes that the step evaluates to multidimensional nodes.

6.1.4 MXPath predicates In MXPath a *predicate* consists of an expression, called a *MXPath predicate expression*, enclosed in square brackets. A predicate serves to filter a sequence, retaining some items and discarding others. Multiple predicates are allowed in MXPath expressions. In the case of multiple adjacent predicates, the predicates are applied from left to right, and the result of applying each predicate serves as the input sequence for the following predicate. For each item in the input sequence, the predicate expression is evaluated and a truth value is returned. The items for which the truth value of the predicate is *true* are retained, while those for which the predicate evaluates to *false* are discarded. The operators (logical operators, comparison operators, etc.) used in MXPath predicates are those used in conventional XPath. MXPath predicates may also contain MXPath expression bodies in the same way as XPath expressions are allowed in conventional XPath predicates. Besides these syntactic constructs, *explicit context qualifiers* (or *ec qualifiers*) are also used in MXPath predicates. An ec qualifier may be applied in every step of a MXPath expression and filter the resulting nodes of the corresponding step according to their explicit context. Explicit context qualifiers are of the form:

```
ec() comparison_op context_specifier_expression
```

The function `ec()` returns the explicit context of the current node. Note that, the predicates assigned to a *context MXPath step* are applied to the context nodes obtained from the evaluation of this step. In the same way, if a MXPath step is a *multidimensional MXPath step*, predicates are applied to the resulting multidimensional nodes.

7 Ordered-Based Context Operations and Comparison

In this section we define how we can apply set operations and comparison among context specifiers when they are represented in Ordered-Based Context Representation.

We first demonstrate how the intersection and union of context specifiers is performed at the level of World Vectors.

Lemma 1. *Let c_1, c_2 be two context specifiers and b_1, b_2 the world vectors of c_1, c_2 respectively. Then the world vector b_3 of the context intersection $c_1 \cap^c c_2$ is obtained by applying the AND operation[3] to the corresponding bits of b_1 and b_2. Respectively, the world vector b_4 of the context union $c_1 \cup^c c_2$ is obtained by applying the OR operation[4] to the corresponding bits of b_1 and b_2.*

Example 7. Consider the context specifiers:
$c_1 = [edition = english]$, and

[3] For this bit-wise AND operation we will use the abbreviation AND_b.
[4] For the bit-wise OR operation we will use the abbreviation OR_b.

$c_2 = [edition = english, customer\_type = student]$.
As we have shown in Example 6 the world vector of the context specifier c_1 is $V(c_1)$=000111. Similarly, it is derived that $V(c_2)$=000100. Then we have:
$b_3 = V(c_1 \cap^c c_2) = 000111\ AND_b\ 000100 = 000100$
and
$b_4 = V(c_1 \cup^c c_2) = 000111\ OR_b\ 000100 = 000111$

It is also possible to compare two context specifiers using their world vectors. This is very useful when we are trying to transform MXML queries containing relevant conditions to SQL queries over a Relational Database. These conditions imply comparisons between the context specifiers which are stored with the MXML document in the relational schema, and the context specifiers which are used in the MXML queries. Similarly to AND_b and OR_b, in Lemma 2 we use the abbreviation XOR_b for the bit-wise XOR operation.

Lemma 2. *Let c_1, c_2 be two context specifiers and b_1, b_2 the world vectors of c_1, c_2 respectively. Then*

1. *$c_1 = c_2$ iff $b_1 = b_2$, alternatively $c_1 = c_2$ iff ($b_1\ XOR_b\ b_2$) = 0*
2. *$c_1 \neq c_2$ iff NOT($b_1 = b_2$)*
3. *$c_1 \geq c_2$ iff ($b_1\ AND_b\ b_2$) = b_2*
4. *$c_1 > c_2$ iff (($b_1\ AND_b\ b_2$) = b_2) and ($b_1 \neq b_2$).*

Example 8. Consider the context specifiers:
$c_1 = [edition = english]$ and
$c_2 = [edition = english, customer\_type = student]$.
Calculating the world vectors of those two context specifiers we have $V(c_1)$=000111=b_1 and $V(c_2)$=000100=b_2. Then the expression $c_1 \geq c_2$ is *true*, as ($b_1\ AND_b\ b_2$) = (000111 AND_b 000100) = 000100 = b_2 (see Case 3 of Lemma 2).

8 Discussion and motivation for future work

Two techniques to store MXML documents in relational databases are presented in this paper. The first one is straightforward and uses a single table to store MXML. The second divides MXML information according to node types in the MXML-graph and, although it is more complex than the first one, it performs better during querying. Additionally, we presented context representation techniques for storing context in a RDB. We also presented MXPath, which is an extension of XPath, in order to query MXML documents and finally, it was shown how we can perform operations and comparisons between context specifiers. Future work will focus on (a) algorithms for SQL translation of MXPath queries giving as the ability for experimental evaluation of the querying performance and (b) optimization of MXML storage using alternative indexing techniques for improving relational schema and query performance.

References

1. T. Amagasa, M. Yoshikawa, and S. Uemura. A Data Model for Temporal XML Documents. In *Proc. of DEXA 2000*, pages 334–344. Springer, 2000.
2. T. Amagasa, M. Yoshikawa, and S. Uemura. Realizing Temporal XML Repositories using Temporal Relational Databases. In *Proc. of the 3rd Int. Symp. on Cooperative Database Systems and Applications, Beijing, China*, pages 63–68, 2001.
3. P. Bohannon, J. Freire, P. Roy, and J. Simon. From XML Schema to Relations: A Cost-Based Approach to XML Storage. In *Proc. of ICDE 2002*.
4. A. Deutsch, M. F. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 431–442. ACM Press, 1999.
5. F. Du, S. Amer-Yahia, and J. Freire. ShreX: Managing XML Documents in Relational Databases. In *Proc. of VLDB' 04*, pages 1297–1300. Morgan Kaufmann.
6. D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDBMS. *Bulletin of the IEEE Comp. Soc. Tech. Com. on Data Eng.*, 22(3):27–34, 1999.
7. M. Gergatsoulis, Y. Stavrakas, and D. Karteris. Incorporating Dimensions in XML and DTD. In Proc. of *DEXA' 01*, LNCS Vol. 2113, pages 646–656. Springer, 2001.
8. M. Gergatsoulis, Y. Stavrakas, D. Karteris, A. Mouzaki, and D. Sterpis. A Web-based System for Handling Multidimensional Information through MXML. In Proc. of *ADBIS' 01*, LNCS, Vol. 2151, pages 352–365. Springer-Verlag, 2001.
9. M. Ramanath, J. Freire, J. R. Haritsa, and P. Roy. Searching for Efficient XML-to-Relational Mappings. In Proc. of *XSym 2003*, pages 19–36. Springer, 2003.
10. J. Shanmugasundaram, E. J. Shekita, J. Kiernan, R. Krishnamurthy, S. Viglas, J. F. Naughton, and I. Tatarinov. A General Technique for Querying XML Documents using a Relational Database System. *SIGMOD Record*, 30(3):20–26, 2001.
11. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In Proc. of *VLDB'99*, pages 302–314. Morgan Kaufmann, 1999.
12. Y. Stavrakas and M. Gergatsoulis. Multidimensional Semistructured Data: Representing Context-Dependent Information on the Web. In Proc. of *CAiSE 2002*, LNCS Vol. 2348, pages 183–199. Springer, 2002.
13. I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proc. of the 2002 ACM SIGMOD Int. Conf. on Management of Data*, pages 204–215. ACM, 2002.
14. W3C CONSORTIUM. XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/, January 2007.
15. F. Tian, D. J. DeWitt, J. Chen, and C. Zhang. The Design and Performance Evaluation of Alternative XML Storage Strategies. *SIGMOD Record*, 31(1):5–10, 2002.
16. F. Wang, X. Zhou, and C. Zaniolo. Using XML to Build Efficient Transaction-Time Temporal Database Systems on Relational Databases. In *Proc. of ICDE 2006*.
17. M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology*, 1(1):110–141, 2001.