Representing the XPath Data Model in Relational Databases

Peter McBrien
pjm@doc.ic.ac.uk

Dept. of Computing, Imperial College, London SW7 2AZ

Abstract. This paper proposes a method of storing XML documents in relational databases which improves upon previous work in three respects. Firstly, it supports all constructs the may appear in an XML document represented in the XPath data model [1], not just elements and attributes. Thus we will achieve a complete representation of XML in relational databases with respect to the XPath data model. Secondly, the method proposed is flexible in how it represents the XML data in a relational database. This flexibility allows one approach to be used to generate schemas that previously would have required the use of different approaches. Thirdly, a set of operators are proposed which restructure the relational schema to produce a more 'natural' relational representation of the data held in XML documents.

1 Introduction

The growing use of the **extensible markup language (XML)** [3] in various Internet related applications has meant that considerable attention has been paid to using XML as a data exchange language between databases, and to the storage of XML in databases.

There are three basic approaches to the storage of an XML document in relational databases. In what we term the **embedded** approach, the entire XML document is stored as a single field in some relational database table. This has the obvious advantage that no consideration need to given to the exact structure of the XML document since it is not being interpreted by the encoding, but also has the obvious disadvantage that no proper use can be made of the database query language to query the XML document. In the **edge schema** approach [6, 8], the XML document is viewed as being a graph structure made up of nodes which can be either elements or attributes, and is held in the database in a table where each row represents an edge in the graph leading to an element or attribute. This approach has the advantage that one schema may be used to hold a variety of XML document types, but has the disadvantage that the querying of the document requires a large number of joins to combine related facts together. In what we term the **element schema** approach [9], separate tables are created for different element names, and fields are created the element's attributes and any child elements that appear at most once. This approach has the advantage

that the relational schema is closer to that which would result from a design process for constructing a relational schema for domain of the data the XML document represents, but has the disadvantage that each XML document type will require a new relational schema to be created.

Neither the edge schema nor element schema approaches handle all features of the XPath data model [1] that is becoming a standard method of specifying queries on XML documents. In particular they ignore the handling of white space and of mixed content, as well as the possible presence of comments and processing instructions withing the XML document. A contribution of this paper is to eliminate this omission for both approaches, by first developing a version of the edge schema capable of supporting all XPath data model constructs, and then showing how that edge schema may be used to create an element schema with similar properties. Another contribution is that it is then shown how a user may ignore certain features of the XML document to build a element schema variant that is a natural representation of the real world domain represented by the XML document type.

Section 2 briefly reviews the XML standard, and the representation of XML in the XPath data model. Section 3 then describes the edge schema and element schema approaches in more detail, and propose how these two representations of XML data may be extended to make the representation complete with respect to the XPath data model, in that the XPath data model may be fully restored from the relational representation of the data. Section 4 proposes a set of operators that may be used to remove certain information from the relational representation of data where that information is not significant to the semantics of the data held. Section 4 also shows that to some extent the use of those operators may be automatically or semi-automatically determined from an XML Schema that validates the XML document being represented in the database.

2 XML and the XPath Data Model

XML **marks-up** data inside pairs of **tags** which are named to indicate the semantics of the data, to form **elements** of the form ⟨tagname⟩data⟨/tagname⟩. Figure 1 illustrates an XML document contrived to show the use of all constructs in the language that may be queried by XPath. The numbers on the left of each line have been added for ease of referencing particular features, and significant whitespace has been shown using printable characters, where new lines are represented by ⏎ and spaces by ␣. This is important, since in XML, ⟨part/⟩ is equivalent to ⟨part⟩⟨/part⟩, but not to ⟨part⟩␣⟨/part⟩. However previous approaches to storing XML in databases always ignore such white space.

The example shows that elements may be nested within other elements, elements may contain text, and elements are allowed to have **attributes**, which are 'name=value' pairs that appear in the opening tag of an element.

XML documents may be stored in any character set, and the **character data (cdata)** that appears inside an element, or as an attribute value, may use any character from that character set, except where the use of that character

```
01 ⟨?xml version="1.0" encoding="UTF-8"?⟩ ↵
02 ⟨ps_db⟩ ↵
03 ␣⟨?sql insert?⟩ ↵
04 ␣⟨part pno="100" colour="red" price="20"/⟩ ↵
05 ␣⟨part pno="101" colour="red" price="22"/⟩ ↵
06 ␣⟨part pno="102" colour="yellow" price="11"/⟩ ↵
07 ␣⟨?sql update?⟩ ↵
08 ␣⟨supplier sno="20I" town="London"⟩⟨!--My␣favourite--⟩ ↵
09 ␣␣⟨name⟩I⟨sup⟩2⟨/sup⟩␣electronics⟨/name⟩ ↵
10 ␣␣⟨supplies⟩100⟨/supplies⟩ ↵
11 ␣␣⟨supplies⟩102⟨/supplies⟩ ↵
12 ␣⟨/supplier⟩ ↵
13 ␣⟨supplier sno="50J" town="Oslo"⟩ ↵
14 ␣␣⟨name⟩J␣&amp;␣S⟨/name⟩ ↵
15 ␣␣⟨supplies⟩100⟨/supplies⟩ ↵
16 ␣␣⟨supplies⟩101⟨/supplies⟩ ↵
17 ␣⟨/supplier⟩ ↵
18 ␣⟨supplier sno="63H" town="Paris"⟩ ↵
19 ␣␣⟨name⟩⟨![CDATA[⟨HTML⟩␣SA]]⟩⟨/name⟩ ↵
20 ␣⟨/supplier⟩ ↵
21 ⟨/ps_db⟩
```

Fig. 1. An XML Document using all XML Constructs significant to XPath queries.

would make the syntax of XML ambiguous. To represent such an ambiguous character, an **entity reference** may be used. For example line 14 represents the & character by &. Alternatively, when there are several ambiguous characters in one text element, a **cdata section** might be more appropriate. This encloses text in an opening ⟨![CDATA[and closing]]⟩, which causes all special characters to be ignored. Hence in line 19 the string '⟨HTML⟩' does not represent an element tag, but instead the sequence of characters shown. However, in the XPath data model, all these types of character data are regarded as representing one type of information, and cannot be distinguished.

In many XML documents, it is the case that each element will fall into one of three categories: (1) those that have nested elements, such as the supplier element in the example which has name and suppliers child elements, (2) those where the element will contain cdata, such as the supplies elements, and (3) those which have no content such as the part elements. However, there is a fourth category called **mixed content** where both non white space cdata and nested elements occur, such as in the name element on line 9. This complicates the representation in a relational database, and the handling of such content has been ignored by previous approaches to storing XML in relational databases.

Apart from data in the form of elements, attributes and character data, XML also allows for additional **meta-data** annotations of the document to be made. Firstly, comments may be inserted anywhere in a document outside of a tag by enclosing text in an opening ⟨!-- and closing --⟩, such as in line 8. XML comments

are meant to be reserved for humans to read, and not as ways on embedding instructions to software as to handle the XML file. For the latter purpose, XML allows for **processing instructions** to be placed inside a document[1]. For example, line 03 may be used to indicate to an program importing the data into an SQL database that the parts data should be inserted, and line 07 might indicate the supplier data should be updated.

2.1 The XPath Data Model for XML

The **XPath** [1] language provides a data model for XML together with a language to identify constructs in an XML document based on constructs in the data model. It is used in a number of XML technologies such as XML Schema [5], XSLT [4] and XQuery [2], and hence may be regarded as the standard data model for XML.

An XML document can be viewed conceptually as a **XPath tree** [8], where comments, elements, attributes, processing instructions, and character data are all types of nodes of the tree. Branches of the tree lead either from the root to a single element, or from an element to any node type. Since the **order** of appearance of everything accept attributes is significant in an XML document, and since attributes appear in a 'special place' inside the element tag, there are two branch types which can be identified: an **attribute branch** type which leads to an attribute node, and an **element branch** type which leads to any node type except attribute nodes (and hence apart from leaf nodes, only leads to element nodes). In common with [8], we draw the tree diagrams with attribute branches leading horizontally out from nodes, and element branches leading vertically from nodes.

Figure 2 illustrates part of the XPath tree representation of the XML document in Figure 1 (the contents of the second two supplier elements are omitted to save space, and are represented by dash lines). Note in particular that the XPath tree explicitly represents the significant white space in the document as cdata nodes. For example, the white space between lines 02 and 03 results in node &2. A common feature of XML processing languages, such as XSLT [4], is to distinguish between processing of XML that ignores white space and processing that does not ignore white space, and we will use a similar notion in Section 4 when we come to describe techniques to improve the relational representation of XML data.

3 Complete Relational Representations of XPath Data

We now modify two well known techniques for the storage of XML in relational databases to provide a complete representation of the XML, in the sense that the

[1] All XML documents may have at their start a pseudo processing instruction called an **XML declaration** similar to shown in line 01 that can specify the version of XML being used, the character encoding, *etc.* Note that the XML declaration only looks like a processing instruction. It is not a processing instruction, and can only appear at the start of a document.

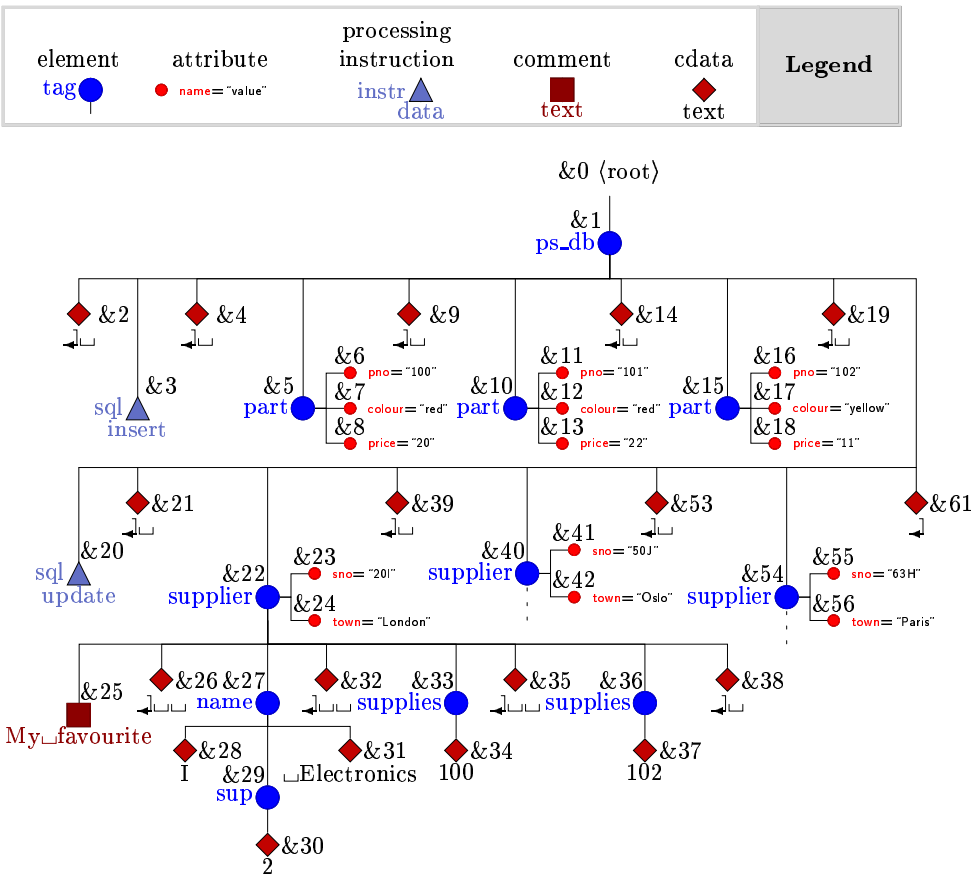

Fig. 2. XPath Data Model annotated with object identifiers

XPath data model may be fully restored from the relational version of the data. The proposed 'canonical' representation of XML in relational databases will be based on the edge schema approach which is detailed next. Then the presentation of data in the element schema is discussed in a manner which clearly relates the element schema representation of data to the edge schema representation of data.

3.1 Edge Schemas

In the **edge schema** approach [6, 9], each XPath edge and associated child node is stored as one table row. Each edge is described by the id of the parent node pid, the order in which it appears under the parent node ord, the name of the child node name, the type of value associated with the node, and the value of the node. In the original version of the edge schema [6] attributes of a parent element were ordered along with child elements, but the canonical XML data model does not order attributes, and they can only appear in the opening tag of

an element. Hence in [9] the edge schema used did not order attributes, which approach is adopted in this paper.

A variant of the edge schema proposed by [6], and used in Monet [8], puts cdata that has different type interpretation into separate tables. For example, supplies contains text in line 10 that would appear to be an integer (which might be made explicit in an XML Schema definition). It follows that the value could be represented as an integer field in a relational database.

It should be noted that none of [6, 8, 9] consider more that just the storage of elements and attributes — *i.e.* comments and processing instructions cannot be represented, white space is assumed to be always ignored and no details of handling mixed-mode content are given. To address these omissions, it is proposed that three new types of row for the edge schema are introduced, giving the following list:

- element nodes use the name field to hold the tag name of the element, and the value field is null. Any cdata or element content of the element will by represented by additional rows of cdata or element type, and these additional rows will have as their pid the nid of the parent element.
- cdata nodes hold the character data in the value field, and leave the name field null.
- attribute nodes use the name and values associated with the attribute to fill in the respective fields in the table.
- process instruction nodes hold the instruction in the name field, and the data of the instruction in the value field.
- comment nodes hold the comment text in the value field, and leave the name field null.

Figure 3 shows the representation of the XML document in Figure 1 in this **XPath complete edge schema**. Note that to save space, the name of element rows is abbreviated to just the leaf element name, rather than the full XPath name. Hence the row with nid=33 represents that element with XPath ps_db/supplier/supplies as the abbreviation supplies.

Note that the edge schema representation has a very direct association with XPath queries [1]. Any step expression in a path expression maps to a select on edge_schema where the type of the step matches the type field in edge_schema, and the parent-child relationship in the XPath query is represented by joins between aliases copies of the edge_schema table.

3.2 Element Schemas

In [9] an alternative approach was proposed, which we refer to here as the **element schema** approach. The element schema approach performs a recursive decent analysis of a DTD schema graph to determine if the table representing an entity should also represent as fields any XML attributes or child elements, or to represent them as separate tables. The method does not deal with the representation of mixed content and assumes white space is always to be ignored. It also does not deal with comments or processing instructions.

edge_schema					
nid	pid	ord	type	name	value
1	0	1	element	ps_db	null
2	1	1	cdata	null	↵␣
3	1	2	process	sql	insert
4	1	3	cdata	null	↵␣
5	1	4	element	part	null
6	5	null	attribute	pno	100
7	5	null	attribute	colour	red
8	5	null	attribute	price	20
9	1	5	cdata	null	↵␣
10	1	6	element	part	null
11	10	null	attribute	pno	101
12	10	null	attribute	colour	red
13	10	null	attribute	price	22
14	1	7	cdata	null	↵␣
15	1	8	element	part	null
16	15	null	attribute	pno	102
17	15	null	attribute	colour	yellow
18	15	null	attribute	price	11
19	1	9	cdata	null	↵␣
20	1	10	process	sql	update
21	1	11	cdata	null	↵␣
22	1	12	element	supplier	null
23	22	null	attribute	sno	20I
24	22	null	attribute	town	London
25	22	1	comment	null	My␣favourite
26	22	2	cdata	null	↵␣␣
27	22	3	element	name	null
28	27	1	cdata	null	I
29	27	2	element	sup	null
30	29	1	cdata	null	2

edge_schema					
nid	pid	ord	type	name	value
31	27	3	cdata	null	␣electronics
32	22	4	cdata	null	↵␣␣
33	22	5	element	supplies	null
34	33	1	cdata	null	100
35	22	6	cdata	null	↵␣␣
36	22	7	element	supplies	null
37	36	1	cdata	null	102
38	22	8	cdata	null	↵␣
39	1	13	cdata	null	↵␣
40	1	14	element	supplier	null
41	40	null	attribute	sno	50J
42	40	null	attribute	town	Oslo
43	40	1	cdata	null	↵␣␣
44	40	2	element	name	null
45	44	1	cdata	null	J␣&␣S
46	40	3	cdata	null	↵␣␣
47	40	4	element	supplies	null
48	47	1	cdata	null	100
49	40	5	cdata	null	↵␣␣
50	40	6	element	supplies	null
51	50	1	cdata	null	101
52	40	7	cdata	null	↵␣
53	1	15	cdata	null	↵␣
54	1	16	element	supplier	null
55	54	null	attribute	sno	63H
56	54	null	attribute	town	Paris
57	54	1	cdata	null	↵␣␣
58	54	2	element	name	null
59	58	1	cdata	null	⟨HTML⟩␣SA
60	54	3	cdata	null	↵␣
61	1	17	cdata	null	↵

Fig. 3. Complete Edge Schema Representation of Data

We will define a **XPath complete element schema** as follows. Each XML element will be represented as a separate table, and have as its fields any attributes of the element, plus a field to represent cdata if there is only one child of the element. and that child is a cdata node (which in XML Schema will correspond to the element being of simpleType). We may build the element schema table @T for XML element with XPath expression @P with attributes @A1,...,@An by the SQL template query elementTableCData in Template 1. If the element is not of simpleType, then we can define a Template 1' for a template called elementTable by omitting the escdata table from the query in Template 1.

Template 1 elementTableCData(@P,@T,@A1,...,@An)

id	parentid	nodetype	localname	prefix	namespaceuri	datatype	prev	text
0	null	1	ps_db	null	null	null	null	null
2	0	7	sql	null	null	null	null	insert
3	0	1	part	null	null	null	2	null
4	3	2	pno	null	null	null	null	null
37	4	3	#text	null	null	null	null	100
5	3	2	colour	null	null	null	null	null
38	5	3	#text	null	null	null	null	red
6	3	2	price	null	null	null	null	null
39	6	3	#text	null	null	null	null	20
7	0	1	part	null	null	null	3	null
8	7	2	pno	null	null	null	null	null
40	8	3	#text	null	null	null	null	101
9	7	2	colour	null	null	null	null	null
41	9	3	#text	null	null	null	null	red
10	7	2	price	null	null	null	null	null
42	10	3	#text	null	null	null	null	22
11	0	1	part	null	null	null	7	null
12	11	2	pno	null	null	null	null	null
43	12	3	#text	null	null	null	null	102
13	11	2	colour	null	null	null	null	null
44	13	3	#text	null	null	null	null	yellow
14	11	2	price	null	null	null	null	null
45	14	3	#text	null	null	null	null	11
15	0	7	sql	null	null	null	11	update
16	0	1	supplier	null	null	null	15	null
17	16	2	sno	null	null	null	null	null
46	17	3	#text	null	null	null	null	20I
18	16	2	town	null	null	null	null	null
47	18	3	#text	null	null	null	null	London
19	16	8	#comment	null	null	null	null	My␣favourite
20	16	1	name	null	null	null	19	null
22	20	3	#text	null	null	null	null	I
21	20	1	sup	null	null	null	22	null
48	21	3	#text	null	null	null	null	2
23	20	3	#text	null	null	null	21	electronics
24	16	1	supplies	null	null	null	20	null
49	24	3	#text	null	null	null	null	100
25	16	1	supplies	null	null	null	24	null
50	25	3	#text	null	null	null	null	102
26	0	1	supplier	null	null	null	16	null
27	26	2	sno	null	null	null	null	null
51	27	3	#text	null	null	null	null	50J
28	26	2	town	null	null	null	null	null
52	28	3	#text	null	null	null	null	Oslo
29	26	1	name	null	null	null	null	null
53	29	3	#text	null	null	null	null	J␣&␣S
30	26	1	supplies	null	null	null	29	null
54	30	3	#text	null	null	null	null	100
31	26	1	supplies	null	null	null	30	null
55	31	3	#text	null	null	null	null	101
32	0	1	supplier	null	null	null	26	null
33	32	2	sno	null	null	null	null	null
56	33	3	#text	null	null	null	null	63H
34	32	2	town	null	null	null	null	null
57	34	3	#text	null	null	null	null	Paris
35	32	1	name	null	null	null	null	null
36	35	4	#cdata-section	null	null	null	null	⟨HTML⟩␣SA

Fig. 4. OpenXML Edge Schema

ps_db

nid	pid	ord
1	0	1

part

nid	pid	ord	pno	colour	price
5	1	4	100	red	20
10	1	6	101	red	22
15	1	8	102	yellow	11

supplier

nid	pid	ord	sno	town
22	1	12	20I	London
40	1	14	50J	Oslo
54	1	16	63H	Paris

supplies

nid	pid	ord	supplies
33	22	5	100
36	22	7	102
47	40	4	100
50	40	6	101

name

nid	pid	ord
27	22	3
44	40	2
58	54	2

sup

nid	pid	ord	sup
29	27	2	2

comment

nid	pid	ord	text
25	22	1	My␣favourite

process_sql

nid	pid	ord	text
3	1	2	insert
20	1	10	update

cdata

nid	pid	ord	text
2	1	1	⌟␣
4	1	3	⌟␣
9	1	5	⌟␣
14	1	7	⌟␣
19	1	9	⌟␣
21	1	11	⌟␣
26	22	2	⌟␣␣
28	27	1	I
31	27	3	␣electronics
32	22	4	⌟␣␣
35	22	6	⌟␣␣
38	22	8	⌟␣
39	1	13	⌟␣
43	40	1	⌟␣␣
45	44	1	J␣&␣S
46	40	3	⌟␣␣
49	40	5	⌟␣␣
52	40	7	⌟␣
53	1	15	⌟␣
57	54	1	⌟␣␣
59	58	1	⟨HTML⟩␣SA
60	54	3	⌟␣

Fig. 5. Complete Element Schema Representation of Data

```
SELECT esp.nid, esp.pid, esp.ord,
       esc1.value AS @A1, ... , escn.value AS @An,
       escdata.value as @T
INTO   @T
FROM   edge_schema AS esp,
       edge_schema AS esc1, ... , edge_schema AS escn,
       edge_schema AS escdata
WHERE  esp.name=@P
AND    esp.nid=esc1.pid AND esc1.type='attribute' AND esc1.name=@A1
AND    ...
AND    esp.nid=escn.pid AND escn.type='attribute' AND escn.name=@An
AND    esp.nid=escdata.pid AND escdata.type='cdata'
```

□

For example, to produce the part table in Figure 5 we would set @T='part', @A1='pno', @A2='colour' and @A3='price', and have no escdata table (since the XML schema definition for part is complexType). To produce the supplies table, we set @T='supplies', have no attributes, but include the escdata table.

To represent cdata which is white space or which appears as part of mixed content, we could either have a separate cdata table for each element path, or as shown in Figure 5 have a single cdata table for all such information. This second approach may be implemented using Template 2 for cData, where the parameter must include all element paths which have not been subject to elementTableCData, which for the example above are ps_db, ps_db/part, ps_db/supplier, and ps_db/supplier/name, giving the cdata table shown in Figure 5.

Template 2 cData(@SIMPLETYPE)
SELECT esc.nid, esc.pid, esc.ord, esc.value
FROM edge_schema AS esp, edge_schema AS esc
INTO cdata
WHERE esp.name IN (@SIMPLETYPE)
AND esp.nid=esc.pid AND esc.type='cdata' □

Comments in the document are handled in a similar way to cdata: either being put in one table per element, or all being put into one table. Template 3 illustrates the later approach.

Template 3 comments()
SELECT esc.nid, esc.pid, esc.ord, esc.value
FROM edge_schema AS esc
INTO comment
WHERE esc.type='comment' □

Processing instructions may either be put all in one table, or put in a separate table for each processing instruction. Template 4 illustrates the later approach.

Template 4 processingInstruction(@I)
SELECT esc.nid, esc.pid, esc.ord, esc.value
FROM edge_schema AS esc
INTO process_@I
WHERE esp.name=@I
AND esc.type='process' □

The complete contents of Figure 5 are specified by the rules used in Example 1. The contents of the table may be automatically be generated from the XML document simply by creating a processInstruction for each processing instruction type in the document, followed by a elementTable or elementTableCData entry for each XPath expression that may match an element, and then declaring in cData all those elements that are subject to elementTable rather than elementTableCData.

Example 1 Specification of Complete Element Schema

```
comment()
processInstruction(sql)
elementTable(ps_db,ps_db)
elementTable(ps_db/part,part,pno,colour,price)
elementTable(ps_db/supplier,supplier,sno,town)
elementTable(ps_db/supplier/name,name)
elementTableCData(ps_db/supplier/sup,sup)
elementTableCData(ps_db/supplier/supplies,supplies)
cData(['ps_db','ps_db/part','ps_db/supplier','ps_db/supplier/name'])
```

□

4 Transformations on the Complete Element Schema

The objective of this section is to propose a set of transformations that may be used to convert the complete element schema of Section 3.2 into a form that can be intuitively regarded as a well designed relational schema. In particular, a set of procedures are proposed which determine which aspects of the XML representation of data are not semantically significant for a particular XML document type, and what action should be taken to remove those irrelevant aspects from the relational representation of the XML document.

When to apply those procedures can be partially determined by any **XML Schema** [5] that might be associated to the XML document. Figure 6 illustrates a possible XML Schema that would validate the XML document in Figure 1, and which will be used to justify some of the examples in the discussion that follows. The processes described have some similarity with the work produced in [7], the difference being that [7] does not address a full XPath data model, and limits its discussion to DTD like semantics in the schema, and therefore is more limited in what it may achieve.

The discussion introduces a series of modification operators, which serve to alter the specification of the templates introduced in the previous section. The modification operators will be illustrated by examples that allow the relational model Figure 7 to be produced directly from the edge schema in Figure 3.

4.1 Handling White space

Often white space is added to XML documents purely for layout purposes in the XML document, and is not intended to have any impact on the semantics of the data. For our example, only the white space that occurs within the name element is significant, since that represents the spaces between words in the textual name of a supplier. In such cases it is natural to revert to the handling of white space in previous approaches — *i.e.* to totally ignore it — for those elements where it is not significant to semantics of the data. The cData template in Template 2 may be called with any element paths for which white space is not to be ignored. Listing no element paths would achieve a similar handling of white space and mixed content as previous approaches. If an XML Schema is available for the

```
22 ⟨xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"⟩
23  ⟨xsd:element name="ps_db"⟩
24   ⟨xsd:complexType⟩
25    ⟨xsd:choice maxOccurs="unbounded"⟩
26     ⟨xsd:element name="part" type="part"/⟩
27     ⟨xsd:element name="supplier" type="supplier"/⟩
28    ⟨/xsd:choice⟩
29   ⟨/xsd:complexType⟩
30   ⟨xsd:key name="part_pk"⟩
31    ⟨xsd:selector xpath="part"/⟩
32    ⟨xsd:field xpath="@pno"/⟩
33   ⟨/xsd:key⟩
34   ⟨xsd:key name="supplier_pk"⟩
35    ⟨xsd:selector xpath="supplier"/⟩
36    ⟨xsd:field xpath="@sno"/⟩
37   ⟨/xsd:key⟩
38   ⟨xsd:keyref name="supplier_supplies_fk" refer="part_pk"⟩
39    ⟨xsd:selector xpath="supplier/supplies"/⟩
40    ⟨xsd:field xpath="."/⟩
41   ⟨/xsd:keyref⟩
42  ⟨/xsd:element⟩
43  ⟨xsd:complexType name="supplier"⟩
44   ⟨xsd:sequence⟩
45    ⟨xsd:element name="name" type="name"/⟩
46    ⟨xsd:element name="supplies" maxOccurs="unbounded" minOccurs="0" type="xsd:integer"/⟩
47   ⟨/xsd:sequence⟩
48   ⟨xsd:attribute name="sno" type="xsd:string" use="required"/⟩
49   ⟨xsd:attribute name="town" type="xsd:string" use="required"/⟩
50  ⟨/xsd:complexType⟩
51  ⟨xsd:complexType name="name" mixed="true"⟩
52   ⟨xsd:choice maxOccurs="unbounded" minOccurs="0"⟩
53    ⟨xsd:element name="sup" type="xsd:string"/⟩
54   ⟨/xsd:choice⟩
55  ⟨/xsd:complexType⟩
56  ⟨xsd:complexType name="part"⟩
57   ⟨xsd:attribute name="pno" type="xsd:integer" use="required"/⟩
58   ⟨xsd:attribute name="colour" type="xsd:string" use="required"/⟩
59   ⟨xsd:attribute name="price" type="xsd:float" use="required"/⟩
60  ⟨/xsd:complexType⟩
61 ⟨/xsd:schema⟩
```

Fig. 6. An XML Schema for the sample XML document

document, an intuitive approach is to remove from the parameter list all element paths which do not have mixed="true" as an attribute of the complexType definition matching the element path.

For Figure 1 and Figure 6 it would be natural to specify the following use of the cData template, since only line 51 for the name element specifies mixed content.

```
cData(['ps_db/supplier/name'])
```

This use of cData would eliminate nodes &2, &4, &9, &14, &19, &21, &26, &32, &35, &38, &39, &43, &46, &49, &52, &53, &57, and &60 from the cdata table in Figure 5.

Often mixed content is used to represent marked-up text, where the elements are mixed inside the character data are present to describe formatting information. In such cases it is sensible to represent the mixed content text from the

XML document directly as text in the database. For our example, the contents of the name element are most naturally represented as a single field (as shown in the supplier table of Figure 7). This is specified using the inline modifier, which due to its recursive decent specification requires a program implementation rather than being implemented by a modification of Templates 1–4.

inline(ps_db/supplier/name)

Note that the use of an element path in inline implies that same path no longer appears in cData.

4.2 Ignoring Irrelevant Information

Processing instructions and comments are not normally part of the semantics of the data, and thus can be ignored in the relational representation. For example, the processing instructions in Figure 1 are clearly related to how to handle the importing of the data into a database, and not facts that need to be held in the database. Also the root node table (ps_db in the example) is only relevant if multiple documents are to be loaded. Any such irrelevant information may be ignored by removing the associated template instruction from the definition of the element schema.

4.3 Identifying Unordered Information

The order of information held may not be semantically relevant. For example, the sequence of parts and suppliers in the document may not be significant, and thus need not be represented in the relational schema. The statement ignoreOrder(⟨element-xpath⟩) means that Template 1 should omit the ord field for any element that matches ⟨element-xpath⟩.

For our example, the following list of ignoreOrder directives can be specified. The first line says that different XML documents of the ps_db need not be ordered. Since the order stored for this element in effect represents the order XML documents are read into the database, this amounts to stating the the order of reading in documents is not significant. For the example XML Schema, the use of choice for the complexType definition in lines 25–28 loosely implies the next two lines, since XML Schema choice normally represents data that is bag of different element types. However this is *not* the semantics of choice: strictly it means that the elements may appear in any order in the document, and not that the order in which the appear is not semantically significant. Similarly, the use of sequence in lines 44–47 implies that order is significant. However, often sequence is used so that different cardinality constraints may be applied to the child elements, since there is no other way to restrict name to just one instance but allow any number of supplies instances under each supplier. Thus in this case it is reasonable to remove the ordering from the name and supplies elements in the fourth and fifth lines below.

```
ignoreOrder(ps_db)
ignoreOrder(ps_db/part)
ignoreOrder(ps_db/supplier)
ignoreOrder(ps_db/supplier/name)
ignoreOrder(ps_db/supplier/supplies)
```

4.4 Identifying Keys

Elements may have some cdata, a child element, or an attribute that is distinct for each occurrence of that element. For example, the pno attribute of part is unique for each instance of part in the example. In this case, the data that is unique may be used in the place of the nid field in Template 1 and also using that value in place of pid in any children of the node.

```
useKey(ps_db/part/@pno)
useKey(ps_db/supplier/@sno)
```

If only one XML document is held in the database then the validating XML schema may be used to extract these useKey definitions from any key statements within the schema (or indeed unique statements with associated mandatory cardinality). For example, the first key above can be derived from lines 30–33, and the second key by lines 34–37. If several XML documents are held in the database, then the above modifiers do not necessary apply, since XML Schema keys do not hold between documents.

4.5 Restructuring

Often a child element of an element appears only once, and contains just cdata. Such child elements are functionally equivalent to attributes of the parent entity, instead of a separate elementTableCData.

```
move(ps_db/supplier/name,../@name)
```

The default SQL type associated with all data is string since that corresponds to the notion in XML that all text between element tags and in attribute types is character data. An informed user, or the contents of an XML Schema, may be used to modify Template 1 to use SQL CAST on those fields to a more specific type. For example, the first modifier below is dictated by line 57 of the XML schema, the second modifier by line 59 that price is a float.

```
setType(ps_db/part/@pno,integer)
setType(ps_db/part/@price,float)
```

If default name given to a relational field derived from the XML document is in appropriate, then it may be renamed. This is simply implemented by altering the name the field is projected as in the appropriate Template definition.

For example, the supplies field of the table supplies in Figure 5 would more appropriately be named pno since it is a foreign key pointing at the pno field in supplier.

rename(ps_db/supplier/supplies/@value,sno)

4.6 Example of an Improved Schema

Modification operators from the previous subsections may be combined together in Example 2 to form an improved relational schema in Figure 7. Note that all except the inline operator can be implemented using slight modified versions of (or just omitting the use of) Templates 1–4.

Example 2 Specification of Improved Element Schema

ignoreOrder(ps_db/part)
useKey(ps_db/part/@pno)
setType(ps_db/part/@pno,integer)
setType(ps_db/part/@price,float)
elementTable(ps_db/part,part,pno,colour,price)
ignoreOrder(ps_db/supplier)
inline(ps_db/supplier/name)
move(ps_db/supplier/name,../@name)
useKey(ps_db/supplier/@sno)
elementTable(ps_db/supplier,supplier,sno,town)
ignoreOrder(ps_db/supplier/supplies)
rename(ps_db/supplier/supplies/@value,sno)
elementTableCData(ps_db/supplier/supplies,supplies)

□

part		
pno	colour	price
100	red	20
101	red	22
102	yellow	11

supplier		
sno	town	name
20I	London	I⟨sup⟩2⟨/sup⟩␣electronics
50J	Oslo	J␣&␣S
63H	Paris	⟨HTML⟩␣SA

supplies	
sno	pno
20I	100
20I	102
50J	100
50J	101

Fig. 7. Improved Element Schema Representation of Data

5 Conclusions

This paper has introduced a complete mapping of XML Documents into a relational database, in the sense that all structure in the XML Document that is significant to the XML XPath data model is maintained in the relational representation of the document. The basic mapping automatically produces from the XML Document a database schema that which is a modification of the edge schema [6]. It has been shown how an XPath complete version of what we term an element schema as used in [9] can be automatically derived from the XPath complete edge schema. Finally we have shown how more 'intuitive' relational schemas may be built by altering the edge schema to element schema mapping using a set of modification rules, and how the use of those rules may in part to determined by the presence of an XML Schema.

The basic edge schema and element schema mappings have been prototyped in a Java application running on the Postgres DBMS (the examples should also work under Sybase and SQL Server), and the prototype is available as part of the DBLibrary package (available from `http://doc.ic.ac.uk/~pjm`). Current work is developing a prototype implementation of the schema modification rules, and extending the improving relational representation by handling candidate keys and foreign keys.

References

1. A. Berglund, S. Boag, D. Chamberlin, M.F. Fernandez, M. Kay, and J. Siméon. XML path language (XPath) 2.0. Technical report, W3C, 2002.
2. S. Boag, D. Chamberlin, M.F. Fernandez, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. Technical report, W3C, 2002.
3. T. Bray, J. Paoli, and C.M. Sperberg-McQueen. Extensible markup language (XML) 1.0. Technical report, W3C, February 1998.
4. J. Clark. XSL transformations (XSLT specification). Technical report, W3C, 1999.
5. D.C. Fallside. XML Schema part 0: Primer. Technical report, W3C, 2001.
6. D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. *Bulletin of the Technical Committee on Data Engineering*, 22(3):27–34, September 1999.
7. M. Mani and D. Lee. XML to relational conversion using theory of regular tree grammers. In *Proceedings of EEXTT and DIWeb 2002*, LNCS, pages 81–103. Springer-Verlag, 2003.
8. A. Schmidt, M.L. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In D. Suciu and G. Vossen, editors, *Proceedings of WebDB2000*, volume 1997 of *LNCS*, pages 137–150, 2001.
9. J. Shanmugasundaram *et al.* A general tecnique for querying XML documents using a relational database system. *SIGMOD Record*, 30(3):20–26, September 2001.