

# On the Persistence of Data Using XML

Is XML a suitable technology for storing complex data?

© Martin Hunter, June 2006

## Executive summary

XML (eXtensible Markup Language) is a text based markup language in which it is possible to represent data and express complex hierarchical relationships between data items. Recently, XML has been a popular tool for the expression, transmission and persistence of a wide range of data structures and sets. This is largely because XML is an open standard, based on simple concepts and principles, is human-readable and can easily and cheaply be shared and persisted using existing protocols and technologies (such as the Internet, HTTP and web servers).

However, the hierarchical nature of XML is limited in its ability to represent complex multi-part associations and data structures. This paper studies a process of normalization of data items in an XML environment and assesses the use of XML as an underlying storage mechanism for applications with complex multi-part relationships between domain entities. It is concluded that XML is inefficient for such purposes.

## Intended audience

This paper is intended for practitioners in computing who are interested in using XML as a means of **persisting** data. It is assumed that readers have a basic understanding of modern programming techniques, including the use of relational data models, hierarchical data models, Object Orientation and a familiarity with the Unified Modeling Language (UML).

## What is XML?

XML is a general-purpose text-based hierarchical markup language in which it is possible to express specific-purpose data elements, their attributes and values in a structured way. XML has been derived from earlier markup languages, specifically SGML, and uses the concepts of *tags* and *attributes* to provide representations of data elements and data fragments. XML is a specification recommended by the W3C (World Wide Web Consortium), and was first developed in the mid 1990s with support from an interest group with more than 150 members.

XML is very similar to look at to other common markup languages in today's computing space, such as Hyper-Text Markup Language (HTML) which is used to express the structure and content of rich formatted pages of text. However, XML differs from HTML in that any data structure can be represented using XML. Usage is not limited to representation of a specific document model.

Whilst it is outside the scope of this paper to define the syntax of XML fully, the following example illustrates a simple example to explain the basics of an XML document:

```
<?xml version="1.0" ?>  
<ELEMENT ATTRIBUTE="123">some text</ELEMENT>
```

In figure 1, the *tag* ELEMENT represents a data element called “ELEMENT”. ELEMENT is defined as containing two items of data:

1. The *named* data item represented by the attribute called “ATTRIBUTE”. In XML, the *value* of an attribute follows its declaration as shown above. There can only be one attribute of any given name within the scope of a particular element.
2. The *unnamed* data item for ELEMENT, captured between the opening and closing *tag statements* of the ELEMENT tag (i.e. <ELEMENT>data</ELEMENT>).

In XML, tags represent specific instances of *types* of data elements. In the example above, we defined an XML document containing one instance of a type of element called ELEMENT.

In order to formally qualify as an XML document, the markup must adhere to the following rules of “well-formed” XML:

1. Only one “root” element exists for a document.
2. All attribute values are quoted with either a single (') or double (") quotes.
3. Tags can be nested but cannot overlap.
4. Non-empty elements (those containing unnamed data) are delimited by opening and closing tags, as follows: <ELEMENT>data</ELEMENT>.
5. Empty elements (those **not** containing unnamed data) can be delimited by a short-hand closing tag, as follows: <ELEMENT />.

## Representing Entities

Whilst XML is more generic than many other data formats in modern computing, it is helpful to draw comparisons with another technique for representing data.

If you have ever worked with relational database (RDBMS) tables, you can imagine the instance of ELEMENT in our document as equivalent to a single *row* from a table called ELEMENT, which has two fields: one called ATTRIBUTE and one which is unnamed. Thus an XML document with multiple instances of ELEMENT would be roughly equivalent to multiple rows from an ELEMENT RDBMS table.

More generally, *element types* in an XML document can be seen to represent logical *entities* in a domain, and instances of these elements represent actual instances of that entity. This is in the same way that an RDBMS table might represent a logical *entity* and the *rows* of data in that table might represent specific instances of that entity.

The following UML class diagram illustrates how XML elements (as defined in an XML schema), RDBMS tables, classes in a class model and logical entities in a domain are all different representations of the same thing, a television:

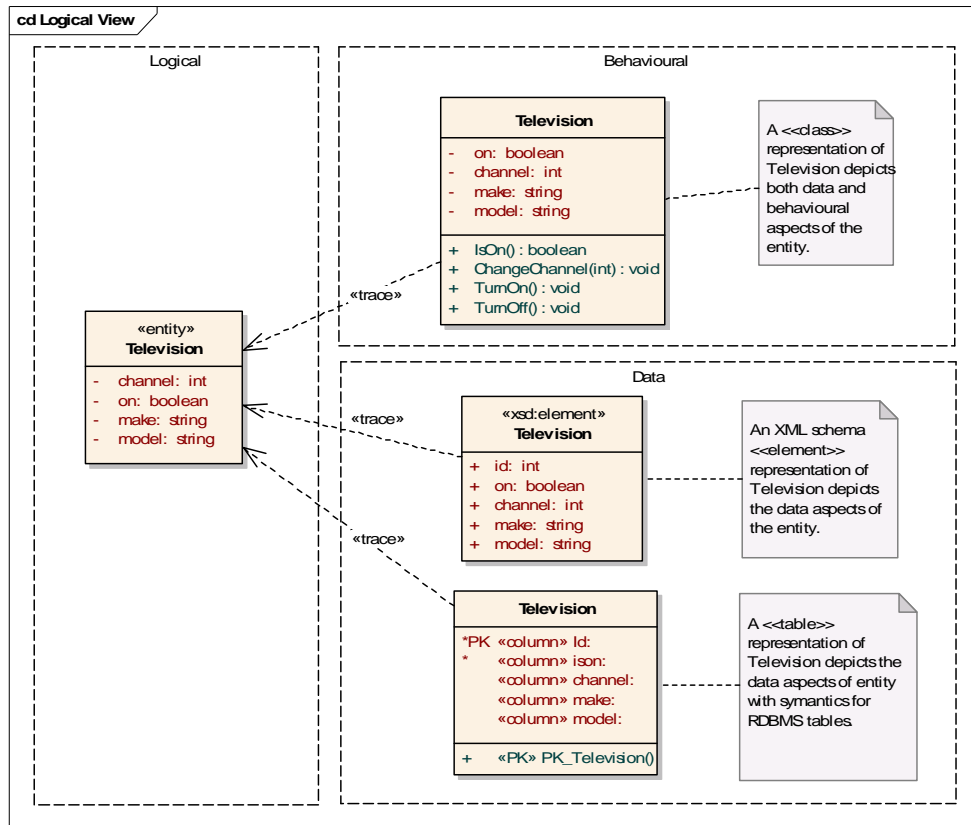


Figure 1: Representing entities

### Representing Relationships

With a RDBMS, we generally have to explicitly define *relationships* between entities through primary and foreign key columns. With XML, relationships can be represented by hierarchical *nesting* of elements as parent-children, as follows.

```
<PARENT-ELEMENT>
  <CHILD-ELEMENT />
</PARENT-ELEMENT>
```

### Using XML to persist data

For all its worth, XML is *just* a markup language. It is a set of tags and attributes that follow a given set of rules (the rules of "well formed" XML) and nothing else. XML is commonly appreciated as a useful way to represent data and *hierarchically structured data*: Useful because it is text based, and therefore easily portable and transmittable, able to represent complex (nested) data structures and is *extensible* (data and data structures in XML documents can be easily extended and reworked). For these reasons, XML has found a particular niche amongst applications working with limited capability text based networks like the World Wide Web and Internet as a medium for data transmission.

However, the very benefits of XML are also a disadvantage. It is a text-based language that uses tags, a mechanism that is not particularly efficient in terms of storage or processing. It is hierarchical and therefore difficult to use when trying to express complex multi-part relationships. Representing such relationships often require the use of value references to abstract the complex relationship *out* of the *nested structure* of XML and into a more appropriate relational form.

## The Problem

Imagine that it has been proposed that serialized XML documents be used to provide an underlying storage mechanism for an application. Thus, data representing entities in the application domain are to be stored in XML documents. Due to the limitations of XML (mainly that it is hierarchical) this may be hazardous. Imagine the following example:

- (1) An XML document,  $xml^A$ , has been created to store data regarding instances of two entities in the application domain - an instance of  $entity^1$  and an instance of  $entity^2$  (*note*: the attribute **id** is used here to uniquely identify the same instance of  $entity^2$ ):

```
<entity1>
  <entity2 id="1" />
</entity1>
```

- (2) A second XML document,  $xml^B$ , is created to store data regarding  $entity^2$  in some other relation with another entity,  $entity^3$ :

```
<entity4>
  <entity3>
    <entity2 id="1" />
  </entity3>
</entity4>
```

In this example the two XML documents,  $xml^A$  and  $xml^B$  contain *duplicate* representations of instance "1" of  $entity^2$ . There is duplicate data in the data storage for the domain. This is inefficient because updates to instance "1" of  $entity^2$  have to be managed in two separate places, thereby introducing a data maintenance issue which requires processing logic to resolve.

## Normalizing the structure

One way to avoid the need for this processing is to abstract out the  $entity^2$  instance "1" into a separate XML document,  $xml^C$ , which is then referenced by the  $entity^1$  and  $entity^3$  instances in  $xml^A$  and  $xml^B$ , as follows:

```
xmlA <entity1>
      <link to="entity2" ref="1" doc="xmlC" />
</entity1>

xmlB <entity4>
      <entity3>
        <link to="entity2" ref="1" doc="xmlC" />
      </entity3>
</entity4>

xmlC <entity2 id="1" />
```

This makes use of a *link entity* in much the same way as a relational database implementation of the same domain model may. In fact, the *link* entity can be abstracted out altogether and be represented by *foreign key* references, as follows, where the attribute "linktoentity<sup>2</sup>" is the foreign key:

```
xmlA <entity1 linktoentity2="1" />

xmlB <entity4><entity3 linktoentity2="1" /></entity4>

xmlC <entity2 id="1" />
```

The structure is now more normal, and duplicate data in storage has been avoided, alleviating the processing overhead mentioned above.

However, what happens if:

- (a) The relationship between entity<sup>2</sup> and either entity<sup>1</sup> or entity<sup>3</sup>, or both, changes? For example, if the relationship between entity<sup>3</sup> and entity<sup>2</sup> is removed, is there still an argument for splitting the relationship between entity<sup>3</sup> and entity<sup>1</sup>, or should they be re-merged into a nested representation of the relationship?
- (b) Another XML document, xml<sup>D</sup>, is introduced which references entity<sup>3</sup>, thus duplicating data representations of entity<sup>3</sup> and potentially tripling representations of entity<sup>2</sup>?

These questions imply two main issues for analysis and design:

- (a) Once the XML documents have been established, it is a large and far reaching task to modify the structure afterwards. The design is *inflexible*, since small changes to the business logic may require larger changes to the structure and number of documents in the system. Modifications of associations between entities can have an impact in terms of requiring a new representation of them but also of modifying the structure of documents already in existence.
- (b) In order to minimize impact due to change, the entities and the associations between entities need to be understood “up front” so that an appropriate document model or split between data representations can be established. In addition, *change cases* needs to be established so as to design-in flexibility in the document system such that it can deal with change as it comes.

### Normalizing further

Following the rules of normalization through, the relationship between all entities are abstracted into reference (link) as opposed to nested representations. This is because that, while nested representations exist, there will always be a potential issue of data duplication, *even within the same XML document*. In the examples above, the document fragments relating to entity<sup>4</sup> and entity<sup>1</sup> could equally be in the *same* XML document.

Normalizing further, we archive the following document, xml<sup>E</sup>, which includes singular representations of each entity instance in the domain without representing relationships through nesting:

```
xmlE <entity1 linktoentity2="1" />
      <entity2 id="1" />
      <entity3 id="3" linktoentity2="1" />
      <entity4 linktoentity3="3" />
```

Notice that this structure is now completely flat in terms of hierarchy, and relies on referential data to deal with associations. This is more flexible and extensible in terms of multi-part associations. However, we now waiver some of the key benefits of using XML to represent data and data structures, such as the ability to use XPath as a querying mechanism and to use nesting to express associations succinctly. We neglect the very benefits of XML due to the restrictions they introduce. As a result, we must now ask *why* are we using XML to represent our entities?

### XML standards for dealing with multiplicity

The XML fraternity recognizes the issues regarding XML's ability to represent complex multi-part relationships. This issue has lead to the development of extensions to XML, such as x:include, which define standard means of representing relationships by *reference* rather than *nesting*.

X:include allows the allocation of a reference to one document fragment or node from another without requiring nodes to be nested. Referenced document fragments are “pulled in” at runtime and the processed “as if” they are nested within the referencing document node. In this manner, querying mechanisms that rely on the nested nature of XML, such as XPath, can be used to query documents that contain multi-part associations that cannot be represented by nesting alone.

X:include is a relatively new technology and many XML parsers on the market today do not offer support for x:include.

### **XML as a view**

It has been argued that the hierarchical, nested nature of XML documents is limiting when it comes to representing more complex data structures in their entirety. However, the hierarchical format is not so limiting when we use it to represent a particular snapshot aspect or *view* of a more complex structure. In the examples above, the nested representations and duplicate data is fine when we think of the XML as being a transient representation of the data structure from a given view, as opposed to a data in a permanent or persistent state.