Principles for an XML Database Engine

Can we structure XML to allow relational data structures and queries?

© Martin Hunter, July 2005

1 Executive Summary

XML has been a popular tool for the expression, transmission and storage 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 stored using existing protocols and technologies (such as the Internet, HTTP and web servers).

Recently a variety of XML based databases have been ushered out to fill an opening in the market around the management, storage and querying of XML based data. However, the limitations of XML mean that many of these databases do not match up to traditional relational databases in terms of expressing data structures and providing high-performance querying and update capabilities.

This paper investigates how we might structure an XML based database to allow the expression of complex relational data structures and queries.

2 Background

2.1 SQL (Structured Query Language)

Most people who use relational databases these days use one form or other of the SQL (Structured Query Language) as a means of expressing query logic. SQL is a mechanism for querying RDBMSs (Relational Database Management System). A standard version of SQL is available, the ANSI SQL, though most RDBMSs implement a SQL command set of their own. This is usually in addition to the ANSI SQL.

SQL is designed to access data models that are governed by a set of relational rules. These rules are usually enforced by a managing application, such as RDBMSs.

2.2 Relational and hierarchical data structures

Hierarchical data structures, such as XML, allow associations between entities in a system to be described in terms of strongly aggregated parent-child relationships. In the XML document in figure 2.2.1, element B is a *child* of A. The relationship between B and A is *strongly aggregated* because the specific <B> element illustrated *cannot exist outside* the specific <A> element illustrated.

```
<A>
    <B/>
</A>
```

Figure 2.2.1

Relational models differ from hierarchical models in that entities in a relational system are *associated*, not *aggregated.* Strongly aggregated entities can be expressed in a relational model, but such restrictions are governed by the logic of the context, and are enforced by a

separate set of contextual *rules*. They are **not** enforced by the format of the data storage mechanism. Figure 2.2.2 shows an entity-relationship model describing a scenario in which A is *associated* with B – but B can be associated with many As, and an A with many Bs (a “many-to-many” relationship).

Figure 2.2.2

This relationship cannot be expressed in terms of a hierarchical model, without duplicating expressions of the *same entity*. In figure 2.2.3, *one* “B” entity (represented by the <B^{1}/> elements) is *associated* with more than one “A” entity:

<A^{1}>
    <B^{1}/>
</A^{1}>
<A^{2}>
    <B^{1}/>
</A^{2}>

Figure 2.2.3

The Xml document above is not properly normalized. Third normal form prescribes that there cannot be more than a single instance of a particular entity in a data model.

It may be convenient to invert this relationship to express A in terms of B (figure 2.2.4), thereby eliminating the duplication of the representation of the entity “B”.

<B^{1}>
    <A^{1}/>
    <A^{2}/>
</B^{1}>

Figure 2.2.4

The inversion in figure 2.2.4 normalises the data structure, but only from the context of the B element. The structure may well, instead, be normalised from the context of the elements A^1 or A^2:

<A^{1}>
    <B^{1}/>
</A^{1}>

or

<A^{2}>
    <B^{1}/>
</A^{2}>

Figure 2.2.5

The structures expressed in figures 2.2.4 and 2.2.5 are therefore not on their own a *full representation* of the data structure, but a *view* of it. Both 2.2.4 or 2.2.5 could be represent a *resultset* obtained from a *query* against the many-to-many relationship, but neither documents that relationship in full. They are insufficient as a means of expressing the proper relationship between A and B.

This issue arises because the principle of *nesting* one element within another can only express one-to-many relationships. It cannot represent many-to-many relationships.

© Martin Hunter, July 2005

In expressing the many-to-many relationship between B and A in figure 2.2.4, we reduce the relationship to a one-to-many, *B is a parent of A*, and therefore *lose* the relationship is that *A is a parent of B*. There is no way, using only this data structure, to infer that A is a parent and B a child. Additional information is required.

The proper parent-child relationship cannot be expressed by nesting, because doing so would lead to a nesting of B within A within B *ad infinatum*. Figure 2.2.6 illustrates:

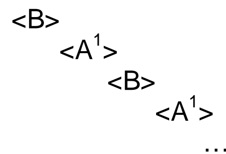

Figure 2.2.6

2.3 XML elements as unique representations of entities

XML elements have *tagnames*, which denote the element's *type*. In figure 2.2.1, there are two *types* of entities: A and B. However, the *actual XML elements* <A> and <B> represent *specific instances* of entities of types A and B. XML documents can contain many identically named elements representing *different* entities of the same type – there can be many <A> elements in a document, for example.

In order to distinguish entities uniquely in an XML document, additional referencing information is required. In figures 2.2.3 – 2.2.6 this has been achieved by marking the elements in question with a numeric identifier. In practice these same effect would be achieved by giving the elements "key" data fields (attributes or sub-elements).

3 Principles for an XML Database Engine

From the arguments documented in section 2, it can be asserted that the discrepancies between the hierarchical data structure that the XML uses, and relational models that the SQL is designed to operate with, give rise to logical issues in the implementation of any XML database that can be queried as if relational.

The following sections discuss some of the possible avenues for developing a "SQL-queryable" XMLDB.

3.1 The problem of unregulated hierarchical structures

Any XMLDB may take the structure of an XML document as *given*. In this context, an externally defined XML-Schema prescribes the format of the XML document used by the XMLDB as a data store, and is *not regulated*. It becomes the job of the XMLDB to provide facilities to query the XML in a structured (in the SQL sense) way.

However, the SQL is a language that is *designed* to operate with relational data structures. For reasons argued above, certain kinds of relational model relationships cannot be expressed in terms of "conventional" XML, without the duplication of *representations of the same entities*, and thus result in de-normalsation of the data structure.

Figure 2.2.3 illustrates a data structure in which an entity "B" is represented more than once in an XML document. It is only by virtue of the fact that we have identified this by naming the "B" elements "<B^1>" that we can see this. There are, in fact, no rules to the XML that distinguish one element as a different representation of the *same entity*. Thus, in pure XML terms, figure

© Martin Hunter, July 2005

2.2.3 might appear as figure 3.1.1. In this context, our XMLDB might reasonably assume that the two <B> elements are representations of *different* “B” entities.

```
<A>
    <B/>
</A>
<A>
    <B/>
</A>
```

Figure 3.1.1

In order to interpret this XML document properly, some additional information is required (the [1]s in figure 2.2.3). This can be undertaken in three broad ways:

(a) Assume that the data structure is normalized to third-normal form.
(b) Regulation of the data structure.
(c) Provision of configuration information regarding the data structure.

In either case, the provision of an XML document *on its own*, it is not a sufficient condition for ensuring that a relational model is followed, and therefore such data structures cannot *alone* be queried using SQL. One has to assume or ensure that a third-normal relational form is adhered to by the XML document.

3.2 Assuming that the data structure is normal

An XMLDB could grant that the documents with which it interacts are *de facto* normalized to, at least, the third normal form. In terms of figures 2.2.3 and 3.1.1, this principle would lead to the treatment of the different <B> elements by XMLDB the as representations of *different* “B” entities in the system.

However, regarding figure 2.2.3 and 3.1.1, this principle would be clearly incorrect. It can be argued that, in fact, no XML document can be trusted to be of a third normal form. Therefore, the assumption that any document will be, is considered misguided.

3.3 A regulated structure

3.3.1 The Principle

Our XMLDB may *regulate* the structure of the XML document with which it interacts to conform to a specific format – follow specific rules – that supports a relational model. The format of the XML data store is thus controlled by the XMLDB.

Though not properly normalized, the following illustrates how a regulated XML document might be structured to store data in relational format:

```
<A>
    <link to=”B” />
    …
    …
</A>
<B>
    <link to=”A” />
    …
    …
</B>
```

Figure 3.3.1

The relationships between A and B are resolved into “link” elements. These elements *describe* the relationships, rather than enforce it via nesting mechanism as in a regular hierarchical data structure as explained in section 2. The relationship information contained within the *nesting* is resolved out and expressed explicitly using a “link” entity.

It is important to note that the <link> elements above are pointers to a *specific* instance of elements. Should more than one element of either type, A or B, exist, further linking information might be required, such as a key or Id reference.

```
<A id=”xyz”>
    <link to=”B” id=”1” />
    …
    …
</A>
<B id=”1”>
    <link to=”A” id=”xyz” />
    …
    …
</B>
```

Figure 3.3.2

In both 3.3.1 and 3.3.2, there is no duplication of the elements representing entities in the system, and therefore the duplication of data can only happen because two *separate* entities *happen* to have identical values (as with properly normalized relational databases).

The provision of a “link” element is only meaningful if the XMLDB interprets it as a “link” entity. Therefore, this model requires the support of configuration information that informs the XMLDB how a link entity is represented in the XML document. It is suggested that an XML-Schema be provided which describes and enforces all such rules that make up a relationally compliant document.

3.3.2 A comparison with RDBMS models

Within RDBMS domain models, relationship data is encoded in a subtly different way, through the use of Primary and Foreign Key fields.

In RDBMS implementations, “Link tables” are often used to resolve many-to-many relationships between entities. RDBMS models rarely use specifically defined “link” entities such as those proposed in section 3.3.1, but the principle is very similar.

The <link> element in figure 3.3.2 is in effect a “link table”. The element is *nested* within an entity element, thus providing information that the link is *associated* with that entity. The link contains explicit information about the *other entity* with which it is associated (the “to” and “id” attributes). Thus, with the nesting and the explicit information combined, the link contains the information required to express the many-to-many relationship.

3.3.3 Normalisation of the <link> element

It can be argued that the document in 3.3.2 is not properly normalised. This is because the link entity (represented by the <link> element) will, for two specific entities that are related to each other, be duplicated in the XML document.

Figure 3.3.3 illustrates. Imagine that the link entity is not represented by a generic element <link>, but by a specific element, <link^1>:

```
<A id=”xyz”>
    <link1>
        <to id=”1”>B</to>
        <to id=”xyz”>A</to>
    </link1>
</A>
<B id=”1”>
    <link1>
        <to id=”1”>B</to>
        <to id=”xyz”>A</to>
    </link1>
</B>
```

Figure 3.3.3

In figure 3.3.3 we can clearly see that the <link> elements in figure 3.3.2 are simply different views of the same *link entity*, and that this link entity is duplicated within the XML document. The <link> element can be resolved out of the elements A and B to normalise the data structure, and avoid duplication:

```
<A id=”xyz” />
<B id=”1” />
<link>
    <to entity=”A”  id=”xyz” />
    <to entity=”B” id=”1” />
</link>
```

Figure 3.3.4

The resolution of the link entity to a single element allows multi-lateral joins between entities to be constructed. Figure 3.3.5 illustrates a tri-lateral join – three entities all related to each other. Such joins can be made between entities of any type; A with A with B for example:

```
<A id="1" />
<A id="abc" />
<B id="2" />
<link>
    <to entity="A" id="1" />
    <to entity="A" id="abc" />
    <to entity="B" id="2" />
</link>
```

Figure 3.3.5

3.3.4 Generalising the XML entity model

All of the entities illustrated so far, except the link entity, have been represented by elements with the tag name of the entity.

The RDBMS domain model in figure 3.3.6 uses a link table “AB” to resolve the many-to-many relationship between A and B. The link table “AB” replaces the element <link> in previous figures:

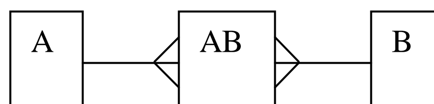

Figure 3.3.6

An XML representation of this might be as in figure 3.3.7:

```
<A ID="1" />
<B ID="2" />
<AB IdA="1" IdB="2" />
```

Figure 3.3.7

A generalised version of this might include entities represented by general <entity> elements that contain “soft” data[1] that describes which entities they related to:

```
<entity type="A" id="1" />
<entity type="B" id="2" />
<link>
    <to type="A" id="1" />
    <to type="B" id="2" />
</link>
```

Figure 3.3.8

3.3.5 Removing unique identifiers

Using a format of <link> elements derived from those in figure 3.3.8, it is possible to link entities together on a spurious set of keys. Entities need not be referred to by a single field designed specifically to provide a unique reference, but on multiple fields. Moreover, the fields used to join, for example, an instance of entity an “A” with an instance of an entity “B” *need not be the same fields* that join a *different* instance of “A” with the same “B”. Figure 3.3.9 illustrates:

```
<entity type="a" blarb="1" />
<entity type="a" foo="abc" />
<entity type="b" blarDeBlar="2" />
<link>
    <to entity="a" on="blarb" keyVal="1" />
    <to entity="a" on="foo" keyVal="abc" />
    <to entity="b" on="blarDeBlar" keyVal="2" />
</link>
```

Figure 3.3.9

3.3.6 A second look at the RDBMS comparison

In section 3.3.3, it is argued that the <link> elements in figure 3.3.2 are duplicated references to the *same link entity*. This argument stands when assessing the nature of the *link entity* with regards the *entity types* to which it relates.

However, a *link entity* is the embodiment of an *association* between entities. A *link entity* is therefore not representative of a “physical” entity, but of their relationship with another entity or set of entities – it is the representation of an *association* that a *single instance of an entity* has with a *set of other entities.*

It follows, therefore, that a link is at most representative of a one-to-many association. This holds even though, on a type-level, entities may be related many-to-many. This implies that the <link> elements in figure 3.3.2 are *not* in fact different views of the same link entity, but represent different link entities: one links A to B, the other B to A.

[1] “soft” data is that data which an element contains which is modifiable without modifying the details of which type of the entity that it represents, in XML terms, such as attribute and element values, as opposed that which is not modifiable (“hard” data) such as the element name or nested position.

3.3.7 Using SQL

The <link> elements in figure 3.3.2 differ from traditional RDBMS domain models in that the relationship between parent and child is recorded and managed by the parent, not the child. In figure 3.3.2, the *parent knows about the child*, since it is the *parent* in which *links* to the child reside. In an RDBMS, the *child knows about the parent*, since it is the child that contains a *foreign-key field* which *refers* to its parent.

In terms of a SQL query for our XML, this poses a problem. Imagine writing a SQL query for the XML document in figure 3.3.10, in which we wanted to obtain all "A"s inner joined on "B"s. The following questions aries:

(a) Which field of B are we inner joining on?

(b) How are we to represent the A to B link entity (marked *) in a SQL statement?

```
<A id="xyz">
    <link to="B" id="1" /> *
</A>
<B id="1">
    <link to="A" id="xyz" />
</B>
```

Figure 3.3.10

To write a SQL statement which inner joins A on B, we need (a) to identify the different *types of links* uniquely with a name, so that they can be referenced in a SQL statement; and (b) identify, for each type of link, the *key field* that the *child entity* is being linked on:

```
<A id="xyz">
    <link to="B" on="id"
name="LINK_TO_B_ON_ID">1</link>
</A>
<B id="1">
    <link to="A" on="id"
name="LINK_TO_A_ON_ID">xyz</link>
</B>
```

Figure 3.3.11

Here, the link has been given a "name" attribute which uniquely identifies its *type*, and "to" and "on" attributes which specifies its type[2]. This is very similar to the model in figure 3.3.9. It is also related to the XML document in figure 3.3.7 and the RDBMS link-table in 3.3.6, in that the link element / table "AB" is *named* to uniquely identify its link type. All that is different here is that the *name* (type) of the link has been resolved out into an attribute value.

SQL statements to inner join the A and B entities might be as follows:

```
SELECT * FROM A, B WHERE A.id = B.LINK_TO_A_ON_ID;
SELECT * FROM A INNER JOIN B ON A.id = B.LINK_TO_A_ON_ID;
```

An alternative approach might be to use the *links* themselves as a substitute table references, and remove the "on" attribute from them:

```
<A id="xyz">
    <link to="B" name="LINK_TO_B" id="1" />
</A>
<B id="1">
```

[2] To normalise this further, the "name" attribute could be used as a *key* to a separate set of link elements which contain the "to" and "on" values, to avoid duplication of the "to" and "on" attributes for links of the same type.

```
    <link to="A" name="LINK_TO_A" id="xyz" />
</B>
```

Figure 3.3.12

A SQL statement to query this might be:

```
SELECT * FROM A, B WHERE A.id = LINK_TO_A.id;
```

However, we are now missing the information which distinguishes the “LINK_TO_A” link as one which is associated with the B entity. There may exist other link entities names “LINK_TO_A” which are associated with entities other than this B. The SQL would have to be clarified, as follows:

```
SELECT * FROM A, B WHERE A.id = B.LINK_TO_A.id;
```

This is now very similar to the SQL, quoted above, which links A to B using a link named “LINK_TO_A_ON_ID”. Though this is not valid SQL, it is more flexible than the approach adopted above since it allows a *single link entity* to register *many* fields on which B can be linked. It does, however, rely on those fields being defined in the link entity, and valid values in those fields. In the extreme, *all* fields of the B entity would be in the *link*, under the guise of “identifying attributes”. Clearly this is nonsensical, as it would lead to massive duplication of data.

We could use the link attributes “to” and “on” as the *unique identifiers* in themselves, thereby removing the need for a *name* to *identify* the type of link. The SQL may be as follows:

```
SELECT * FROM A, B WHERE A.id = B.id;
```

There is a logical confusion that now arises: the term “B” refers to both an *actual* entity (in the FROM clause) and a *link* entity (in the WHERE clause).

The code B.id above refers to the “id” attribute of the B type entities. However, we know that we are actually trying to refer to the “id” attribute of the link entity named LINK_TO_A_ON_ID. A processor of this SQL would have difficulty distinguishing the meaning of this SQL, and would therefore have to make assumptions about it. To process this, for example, we may make the assumption that the first term in the equation “A.id = B.id” (“A.id”) is the *parent* (refers to an *actual* entity) and the second term (“B.id”) is the *child* (refers to a *link* entity). Clearly this may be contradictory to the intent of the user that constructed this SQL, who may genuinely want to construct a resultset which returns a join between A and B on their respective *actual* “id” attribute values.

Clearly, it is important to distinguish *link* entities from *actual* entities by *naming* them.

3.4 Provision of configuration information

An XMLDB could be provided with configuration information that is specific to the data structure with which it interacts. This configuration information could present the *rules* that let the XMLDB know how to treat relationships within the data document (for example, the relationship between A and B in figure 2.2.3).

This differs from the solution proposed in section 3.3, since the provision of configuration information for the XMLDB to use is the responsibility of the party providing the data XML document. In section 3.3, the XMLDB *prescribes* the configuration.

The configuration information could be provided in a number of ways:

(a) A separate configuration file could be provided, like a config. file, XML document or XML-Schema, which documents these rules. This can be used in conjunction with any XML-Schema describing the basic rules of relational data structures.

(b) The XML document could be treated as an “original source” document, which could be transformed into a format which can be treated as a relational store, using something like XSLT. Again, this can be used in conjunction with any XML-Schema describing the rules of relational data structures.

(c) The configuration can be hard-wired into the XMLDB engine, though clearly this would preclude any use of the engine with different data formats / documents.