0.1 XML Storage techniques

There are several techniques for storing XML documents in relational databases that could be employed in this project. Some of the more common ones are discussed here. The techniques will be evaluated against two criteria:

1. all XML document structural information must be retained so that the document can be regenerated.
2. The technique must store the data in an intuitive form so that the user can manipulate the data with similar ease to that of a conventional relational database.

The following example XML document will be used when describing the techniques:

```
<ps_db>
      <part pno="100" colour="red" price="20"/>
      <part pno="101" colour="red" price="22"/>
      <part pno="102" colour="yellow" price="11"/>
      <supplier sno="20I" town="London">
            <name>
                  I
                  <sup>2</sup>
                  electronics
            </name>
            <supplies>100</supplies>
            <supplies>102</supplies>
      </supplier>
      <supplier sno="50J" town="Oslo">
            <name>J &amp; S</name>
            <supplies>100</supplies>
            <supplies>101</supplies>
      </supplier>
      <supplier sno="63H" town="Paris">
            <name>
                  <![CDATA[<HTML> SA]]>
            </name>
      </supplier>
</ps_db>
```

0.1.1 Edge Storage

Many variations of relational storage of XML focus on storing information about the edges between elements in an XML document. [4] suggests 3 related approaches.

0.1.1.1 Edge Approach

Two tables are created: an edge table and a value table. The value table stores all attribute values and character data values. These are referenced by a row in the edge table, where each row represents an edge between a parent and child element. Separate value tables can be created for different types of data (integer, string, id reference).

Edge				
source	ordinal	name	flag	target
1	1			
1	2			
1	3			
1	4			
1	5			
1	6			
2	1			
2	2			
2	3			
3				
3				
3				
4				
4				
4				
5	1			
5	2			
5	3			
6	1			
6	2			
6	3			
7	1			
8	1			

Edge source ordinal name flag target 1 1 part ref 1 1 2 part ref 2 1 3 part ref 3 1 4 supplier ref 4 1 5 supplier ref 5 1 6 supplier ref 6 2 1 pno decimal v1 2 2 colour string v2 2 3 price decimal v3 3 1 pno decimal v4 3 2 colour string v5 3 3 price decimal v6 4 1 pno decimal v7 4 2 colour string v8 4 3 price decimal v9 5 1 name ref 8 5 2 supplies decimal v10 5 3 supplies decimal v11 6 1 name string v12 6 2 supplies decimal v13 6 3 supplies decimal v14 7 1 name string v15 8 1 sup decimal v16 vdecimal vid value v1 100 v4 101 v7 102 v10 100 v11 102 v13 100 v14 101 v16 2 v3 20 v6 22 v9 11

vstring vid value v2 red v5 red v8 yellow v12 J & S v15 <HTML> SA

0.1.1.2 Binary Approach

A separate table is created for each element or attribute name. Each row in any table indicates an edge from that element instance to another element.

Bps_db source ord vdecimal vstring target null 1 null null 1 Bsupplier source ord vdecimal vstring target 1 4 null null 5 1 5 null null 6 1 6 null null 7

Bpart source ord vdecimal vstring target 1 1 null null 2 1 2 null null 3 1 3 null null 4 Bsno source ord vdecimal vstring target 5 1 null 20I null 6 1 null 50J null 7 1 null 63H null

Bpno source ord vdecimal vstring target 1 1 100 null null 2 1 101 null null 3 1 102 null null Btown source ord vdecimal vstring target 5 2 null London null 6 2 null Oslo null 7 2 null Paris null

Bcolour source ord vdecimal vstring target 1 2 null red null 2 2 null red null 3 2 null yellow null Bname source ord vdecimal vstring target 5 3 null null 8 6 3 null J & S null 7 3 null <HTML> SA null

Bprice source ord vdecimal vstring target 1 3 20 null null 2 3 22 null null 3 3 11 null null Bsup source ord vdecimal vstring target 8 1 2 null null

Bsupplies source ord vdecimal vstring target 5 4 100 null null 5 5 102 null null 6 4 100 null null 6 5 101 null null

0.1.1.3 Universal Table

A single table is created to store all information.

source ord flag targetsupplier ord flag targetsno ord flag targetsupplies . 1 4 ref 2 1 string 20I 1 decimal 100 1 4 ref 2 1 string 20I 2 decimal 102 1 5 ref 3 1 string 50J 1 decimal 100 1 5 ref 3 1 string 50J 2 decimal 101 1 6 ref 4 1 string 63H null null null

The number of columns in the table is the three times the total number of element and names (each of which has an ord, flag and target column). Only a subset of the columns and the rows have been shown here.

All approaches satisfy criteria (b) of a lossless transformation from an XML document to a relational data representation.

The problem with the first approach is that the value of an attribute or element is completely alienated from its name (the name and value pair are distributed across two relations) which certainly doesnt satisfy criteria (a). The next two resolve this problem, but as a result lead to denormalization: for example, the data of an element with two children is repeated in the two rows where the edge is represented. Also, there are many null values which are undesirable for many reasons. Denormalization is problematic when updating data (multiple updates are required instead of just 1).

0.1.1.4 Edge Schema

Another edge representation is described in [3,9].

edge schema id pid ord type name value 1 0 1 element ps_db null 2 1 1 element part null 3 1 2 element part null 4 1 3 element part null 5 1 4 element supplier null 6 1 5 element supplier null 7 1 6 element supplier null 8 2 1 attribute pno 100 9 2 2 attribute colour red 10 2 3 attribute price 20 11 3 1 attribute pno 101 12 3 2 attribute colour red 13 3 3 attribute price 22 14 4 1 attribute pno 102 15 4 2 attribute colour yellow 16 4 3 attribute price 11 17 5 1 attribute sno 20I 18 5 2 attribute town London 19 6 1 attribute sno 50J 20 6 2 attribute town Oslo 21 7 1 attribute sno 63H 22 7 2 attribute town Paris 23 5 1 element name I * electronics 24 5 2 element supplies 100 25 5 3 element supplies 102 26 6 1 element name J & S 27 6 2 element supplies 100 28 6 3 element supplies 101 29 7 1 element name <HTML> SA 30 23 1 element sup 2

This is an improvement on the aforementioned because there is less denormalization. There is less denormalization because rather than have 3 columns with only one ever containing applicable data there are 2 equivalent columns, with the first saying how to interpret the data in the second. There is still denormalization because rows with type value 'element' contain no information in the 'value' column.

Until the database has been normalized sufficiently, it cannot possibly be considered well-designed and intuitive to a database user.

To reduce denormalization to an acceptable level, the relational schema must be generated to suit the underlying schema of an XML document. In the edge approaches, the only XML schema information used was that which was self-contained in the markup. But a particular XML document seldom reveals the whole schema. A representation of the whole schema is required.

0.2 XML Schemas

Several XML schema-specification languages exist. They include:

- Document Type Definition (DTD)
- XSD Schema (XML Schema Definition)
- XDR (XML-Data Reduced)
- SOX (Schema for Object-Oriented XML)
- Schematron § DSD

DTD and XSD Schema are by far the most common. Of the two, DTD has been used more widely thus far due to its relative longer existence and greater compactness.

However, XSD Schema is likely to replace DTD particularly in the more serious applications. There are many reasons for increasing preference to XSD Schema:

- It supports object-oriented structuring of data. Type definitions can be extended.
- It supports data types. Content validation is thus more rigorous.
- It is written in XML syntax. Many of the powerful XML parsers are therefore usable with XSD Schema documents as well.

Thus we shall only study relational schema generation from XSD Schema.

The XML document introduced earlier conforms to the following XSD Schema:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="ps_db">
        <xsd:complexType>
            <xsd:choice maxOccurs="unbounded">
                <xsd:element name="part" type="part"/>
                <xsd:element name="supplier" type="supplier"/>
            </xsd:choice>
        </xsd:complexType>
        <xsd:key name="part_pk">
            <xsd:selector xpath="part"/>
            <xsd:field xpath="@pno"/>
        </xsd:key>
        <xsd:key name="supplier_pk">
        <xsd:selector xpath="supplier"/>
            <xsd:field xpath="@sno"/>
        </xsd:key>
        <xsd:keyref name="supplier_supplies_fk" refer="part_pk">
            <xsd:selector xpath="supplier/supplies"/>
            <xsd:field xpath="."/>
        </xsd:keyref>
    </xsd:element>
    <xsd:complexType name="supplier">
        <xsd:sequence>
            <xsd:element name="name" type="name"/>
                  <xsd:element name="supplies" type="xsd:integer"
minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="sno" type="xsd:string" use="required"/>
        <xsd:attribute name="town" type="xsd:string" use="required"/>
    </xsd:complexType>
    <xsd:complexType name="name" mixed="true">
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="sup" type="xsd:string"/>
        </xsd:choice>
    </xsd:complexType>
    <xsd:complexType name="part">
        <xsd:attribute name="pno" type="xsd:integer" use="required"/>
        <xsd:attribute name="colour" type="xsd:string" use="required"/>
        <xsd:attribute name="price" type="xsd:float" use="required"/>
    </xsd:complexType>
</xsd:schema>
```

0.2.1 Schema-Adaptive XML Storage

[3,9] explain another approach which does make use of schema information. [9] has called it the Element Schema. It creates separate relations for the different

types of elements; any attributes they contain become attributes of the relation[1]. The information about different types of elements is obtained from the XML schema.

ps_db id pid ord 1 0 0 name id pid ord name 8 5 1 I * electronics 9 6 1 J & S 10 7 1 <HTML> SA

part id pid ord pno colour price 2 1 1 100 red 20 3 1 2 101 red 22 4 1 3 102 yellow 11 supplies id pid ord supplies 11 5 1 100 12 5 2 102 13 6 1 100 14 6 2 101

supplier id pid ord sno town 5 1 4 20I London 6 1 5 50J Oslo 7 1 6 63H Paris sup id pid ord sup 15 8 1 2

This relational schema is normalized to an acceptable degree. It will be the basis for our storage method.

However, several problems remain:

- The element schema relations still contain noise that is of no use to a conventional database user, impeding their ability to browse the dataset. This noise data is only required to reconstruct the shredded XML document.
- This noise doesnt stop short of being a nuisance: it actually causes a serious problem when overlapping documents are stored (see below for explanation).
- Simple elements are stored in a separate relation (e.g. 'sup') when theyd more intuitively be represented as columns in the parent elements table.

'Noise' Columns There hasnt been much work addressing (a). One reason is that the aim of existing work such as [3] is not to provide XML support in a conventional relational database. Rather, it is to store XML documents in the efficient relational structure but to be used in a Native XML Database. A Native XML Database is a database that is queried using XML query languages (such as XPath and XQuery). Logically, the database is one large XML file made up of the data from several smaller documents (only physically is it relational). Inserted and retrieved data are presented in XML format. While XML Databases have exciting potential, trust remains with relational databases and is likely to for some time. The aim of this project is not to build an XML Database, but to incorporate support for XML seamlessly in a relational database for conventional use.

This project is to address the problem of noise columns.

Data 'Collision' Problem (b) is not immediately obvious, but is a very serious one. Imagine a very simple scenario where the data from two documents is imported into the database.

[1]We use the term 'column' when referring to relational attributes to avoid confusion with XML attributes

```
<ps_db>
        <part pno=”100” colour=red price=”20”/>
</ps_db>
<ps_db>
        <part pno=”100” colour=red price=”30”/>
</ps_db>
```

These documents contain information regarding the same physical entity (namely, the red part given the number ’100’). The intention of importing the second document is to update the price of the part. However, the element schema representation of these documents would be:

part id pid ord pno colour price 2 1 1 100 red 20 4 3 1 100 red 30

The data from the first document remains intact without any price change. How is a conventional database user supposed to determine the price of the part with number 100? The user only expects to see one relevant row. Furthermore, when the first document is regenerated, it will not contain the latest price of this part.

The reason that outdated information is retained is that the primary key is the id column. Regardless of the real-world equivalence between the ’two’ parts, they are treated completely independently in this relational schema because they originate from different documents. Again, this is a consequence of the denormalized relational schema: the price value is uniquely identified by the value of a column not declared as a key attribute. The problem is *not* solved by making ’pno’ the primary key instead and thus only allowing one row containing this pno value. This is because structural information about the first document will be overwritten. It is also not solved by making id *and* pno primary key attributes, because two rows will still exist for the same part.

The ’Element Schema’ was only intended to provide a relational storage for XML documents, without any concern for overall data consistency. It is left for this project to address this problem too.

Simple Elements in separate tables The element schema interprets the schema of the XML document rather literally with limited regard for the deeper meaning.

What is required is a more abstract, conceptual interpretation of the XML schema that resembles a human users interpretation more closely.

As stated, problems (a) and (b) are left for this project to address. Problem (c) can be addressed by investigating further work.

The Constraints Preserving and Inlining (CPI) algorithm described in [4] (and originally proposed in [21]) deals with problems such as (c) by developing a *conceptual model* of the XML schema. The conceptual model used is the DTD Graph. From the DTD Graph, a relational schema is built. Because the DTD Graph resembles a human user’s interpretation of a schema more closely, the relational schema is likely to be more intuitively structured.

However, it has already been decided that XSD Schema is the XML schema of choice. Therefore, the CPI algorithms specific operation cannot be adopted in this project. Despite this, its general *strategy* can be. This strategy to generate a relational schema from an XML schema is to:

- create a conceptual model of the schema
- build a relational schema that reflects this conceptual model

We investigate two conceptual modelling techniques for XSD Schemas.

0.2.1.1 Conceptual Modelling of XSD Schemas

One conceptual modelling technique for XSD Schema is given in [14] - the X-Entity model.

The constructs are defined as follows:

- rectangle - an entity (from a complex element)
- diamond - a relationship between entities (from containment of one element within another)
- curved bisector a disjunction (one of the containment relationships it bisects may be instantiated)
- oval - an attribute (multi-occurring attributes have a double oval). The line connecting the entity and the attribute is solid if the attribute is required and dashed if it is optional. Primary key attributes are underlined.

However, in attempting to achieve generality among XML schemas (i.e. it can model DTDs and XSD Schemas), it has neglected to represent some fundamental aspects of XSD Schema. For example, it does not distinguish between entities originating from elements with and without mixed content. It also does not distinguish between attributes originating from XML elements and those originating from XML attributes. Schema information is being lost during the modelling process, so this model shall not be used.

We now consider the conceptual model proposed in [20] - the Extensible Entity Relationship Model. While this can also apply to DTDs, it provides a wider range of constructs to cater for those that can arise from XSD Schemas. It also resembles the conceptual model of the target schema (the relational schema) relatively closely and thus conversion will be more straightforward. This is the definitive feature that makes it a strong choice for use in our relational schema generation algorithm. Indeed this is the modelling technique that shall be employed. It is described in detail below.

No mechanisms are proposed for mapping the model to a relational schema. This is left for our project to investigate.

Note that hereafter 'XSD Schema' will be referred to by its more familiar name, 'XML Schema'.

0.3 Extensible Entity Relationship Modelling

First the basic constructs are introduced. Then their derivation from an XSD Schema is described.

0.3.1 Basic Constructs

The Extensible Entity relationship model shares many features with its more familiar predecessor, the Entity relationship model. It is based on:

- entities
- attributes
- relationships.

0.3.1.1 XER Entity

An entity is displayed as a rectangle as expected, with the top label being the name given to it. Its attributes are ordered, and is therefore called an **Ordered Entity**.

There may be entities with no attribute ordering specified. Such an entity is denoted by a rectangle with a pointed base[2] . It represents an **Unordered Entity**.

There may also be entities with content occurring outside any of the entitys child constructs, yet nonetheless part of the entity. Such an entities is called a **Mixed Entity**. It is represented by a rounded box instead of a rectangle.

0.3.1.2 XER Attribute

Just like entities, attributes occur in different flavours to account for the fact that they can originate from related but quite distinct constructs in XML: an attribute and an element that has a simple type (i.e. it only contains character data).

To incorporate such information, XER attributes originating from XML attributes are prefixed with an '@' symbol. Those that originate from an element have no prefix.

As with ER attributes, primary key attributes in an XER Entity are underlined. Multi-valued attributes are suffixed by a '*'.

0.3.1.3 XER Generalization

Some constructs may be categorized by some common 'super entity'. This super entity is modelled as a Generalization.

The specialization constructs are not limited to entities; they can be XER attributes too.

[2]This is not the representation used in [20],which looks exactly like an ordered entity but prefixes the name with a '?'. This becomes a problematic when illustrating mappings in later sections, and so a differently shaped construct will be used instead.

0.3.1.4 XER Relationships

Relationships are displayed in exactly the same format as in an ER model. A line with a diamond in the middle (with the name of the relationship inside the diamond) connects the two related entities. The cardinality constraints are specified as expected.

0.3.2 Building the XER Model from an XML Schema

To illustrate all aspects of the modelling, the mappings will be explained using modified versions of the constructs that occur in the parts and suppliers XSD Schema introduced earlier. Miscellaneous examples are occasionally used where no suitable construct exists in the parts and suppliers schema. The full XER model of the parts and suppliers schema is shown at the end.

Before continuing, it should be noted that [20] does not illustrate how to model every possible XML Schema construct. It only considers: § elements (simple, complex and mixed) § attributes § containment of complex elements § model groups (all, sequence, choice)

[20] does not discuss some constructs. These include key, keyref and group. However, modelling of these will be discussed here.

There are also some more complex contexts in which the constructs can occur which [20] does not address. These will be considered here.

Other constructs such as those which support inheritance will not be considered. Elements and Attributes An element with a complex type maps to an XER entity. In the example given, the element part becomes an XER entity named part.

The schema attribute inside the complex type definition becomes an XER attribute of the entity. Thus, the price, pno and colour attributes become XER attributes of the part entity, and are prefixed with an @ symbol.

<xsd:element name="part"> <xsd:complexType> <xsd:attribute name="pno" type="xsd:integer" use="required"/> <xsd:attribute name="colour" type="xsd:string" use="optional"/> <xsd:attribute name="price" type="xsd:float" use="required"/> </xsd:complexType></xsd:element>

The usage (required or optional) is denoted in the expected way (? meaning optional, nothing meaning required).

If the complex type definition is globally accessible (rather than defined inline), the entity should take the name of the type, not the element instance.

A simple element within a complex element can be modelled very similarly, but unlike XML attributes, elements can occur multiple times. The cardinality of the simple element is determined by its maxOccurs attribute value. If it is greater than 1, if becomes a multi-valued attribute.

<xsd:element name="supplier"> <xsd:complexType> <xsd:sequence> <xsd:element name=supplies maxOccurs=1> </xsd:sequence> </xsd:complexType></xsd:element>

A complex element which can contain mixed content is modelled as an XER Mixed Entity.

<xsd:element name="paragraph"> <xsd:complexType mixed=true> <xsd:attribute name=bold/> <xsd:attribute name=italic/> </xsd:complexType></xsd:element>

A complex element within another complex element means that there will be two separate entities (one per element), with the outer element referencing the inner one via an XER Relationship.

<xsd:element name=book> <xsd:complexType> <xsd:element name=chapter minOccurs=1 maxOccurs=unbounded/> </xsd:complexType></xsd:element >

Figure x: Mapping nested schema elements (which are complex) to XER Relationships

There are some points to note regarding cardinalities. The minOccurs and maxOccurs attributes correspond as expected in the XER Model (and when none are specified XML Schemas convention is that they are implicitly both 1).

More interestingly, there doesnt appear to be any information about the cardinality in the reverse direction in the XML Schema. However, XMLs data model is known to be tree-like. By definition, this means that child entities can belong to one and only one parent. Thus it can be inferred that the cardinality at the childs end must be 1:1. This inference is only true for containment relationships (i.e. when a complex element is contained within a complex element). Relationships that do not arise from containment are handled differently as shall be seen shortly.

Keys and Keyrefs

Within a complex type definition a unique identifier may be specified via the key construct. The fields which are declared as constituents of the key are modelled as XER primary key attributes within the entity that corresponds to the complex type definition.

<xsd:element name=book> <xsd:complexType> <xsd:key name=book_pk> <xsd:selector xpath=book/> <xsd:field xpath=@ISBN/> </xsd:key> </xsd:complexType></xsd:element >

Figure x

A reference from one complex element to another is achieved via a keyref tag. This corresponds to an XER Relationship. <xsd:element name=book> <xsd:complexType> <xsd:keyref name=supplied_by refer=publisher_pkey> <xsd:selector xpath=publisher/> <xsd:field xpath=@publisherName/> </xsd:key> </xsd:complexType></xsd:element >

Figure x: Mapping between schema sequence and XER Entity (Note: for simplicity the intermediate entity needed for the sequence element has been omitted) However, unlike previously, the referred entity can exist independently of the referring entity. Thus the cardinality at the referred entitys end is zero to n. The cardinalities at the referring entitys end will depend on the characteristics of the attributes concerned (for example, if an attribute in a keyref field was optional then the relationships lower bound would be 0 whereas if it was required then it would be 1 etc.).

Model Groups

Within the complex type definition a model group may be specified (the schema constructs all, choice, group and sequence are all model groups).

As was seen from the multi-valued attribute example, a sequence simply means all its child constructs are added to the entity. A group is also modelled in this way.

A choice construct means that only one of its child elements may occur. In such a case it becomes necessary to use an XER Generalisation.

<xsd:element name=ps_db> <xsd:complexType> <xsd:choice> <xsd:element name=part/> <xsd:element name=supplier/> </xsd:choice> </xsd:complexType></xsd:element >

Figure x: Mapping between schema choice and XER Generalization

The all construct means that its contents can appear in any order, and so would be modelled by an unordered entity.

<xsd:element name=book> <xsd:complexType> <xsd:all> <xsd:attribute name=priceUSD/> <xsd:attribute name=priceGBP/> </xsd:all> </xsd:complexType></xsd:element >

Figure x: Mapping between schema all and XER Unordered Entity

More complex model group constructs are possible. For example: § a model group may have a maxOccurs value greater than 1 § attributes can occur outside a model group but nonetheless inside the complex type definition containing the model group. § model groups may be nested

[20] doesnt consider such contexts, but the parts and supplier schema does indeed contain such contexts and these must be handled. What is done is to consider a model group as another entity, with a relationship from the original entity. This allows multiple instances to exist, outer attributes to be kept separate and nesting via a series of relationships.

<xsd:element name=name> <xsd:complexType> <xsd:attribute name=text type=xsd:string/> <xsd:choice minOccurs=0 maxOccurs=unbounded> <xsd:element name=sup type=xsd:string/> </xsd:choice> </xsd:complexType></xsd:element >

Figure x

A Complete Example The modelling techniques above have been shown on some miscellaneous examples as the original example (the parts and suppliers schema) is not diverse enough to illustrate all constructs.

The complete XER model of the parts and suppliers schema would look as follows:

One may point out that the supplies attribute is not shown. This is intentional because, like with ER models, foreign key attributes are not explicitly shown in the entity as they are implied by the relationship.

As stated, the task that remains is to translate the model into a relational schema.